Minimizing Computation in Convolutional Neural Networks

Jason Cong and Bingjun Xiao

Computer Science Department,
University of California,
Los Angeles, CA 90095, USA
{cong,xiao}@cs.ucla.edu

Abstract. Convolutional Neural Networks (CNNs) have been successfully used for many computer vision applications. It would be beneficial to these applications if the computational workload of CNNs could be reduced. In this work we analyze the linear algebraic properties of CNNs and propose an algorithmic modification to reduce their computational workload. An up to a 47% reduction can be achieved without any change in the image recognition results or the addition of any hardware accelerators.

1 Introduction

Biologically inspired convolutional neural networks (CNNs) have achieved good success in computer vision applications, e.g., the recognition of handwritten digits [7, 10], and the detection of faces [2, 8]. In the 2012 ImageNet contest [1], a CNN-based approach named SuperVision [6] outperformed all the other traditional image recognition algorithms. On one hand, CNNs keep the advantage of artificial neural networks which use a massive network of neurons and synapses to automatically extract features from data. On the other hand, CNNs further customize their synapse topologies for computer vision applications to exploit the feature locality in image data.

The success of CNNs promises wide use for many future platforms to recognize images, e.g., micro-robots, portable devices, and image search engines in data centers. It will be beneficial to improve the implementation of the CNN algorithm to reduce computational cost. One direction is to improve the CNN algorithm using hardware accelerators, e.g., GPUs and field-programmable gate arrays (FPGAs) [3,5,9]. Another orthogonal direction is to reduce the theoretical number of basic operations needed in the CNN computation from the algorithmic aspect, as will be discussed in this work. Here, we first reveal the linear algebraic properties in the CNN computation, and based on these properties, we propose an efficient algorithm that can be applied to generic CNN architectures to reduce the computational workload without any penalty on the image recognition quality or hardware cost.

2 Background

2.1 Algorithm Review of CNNs

Convolutional neural networks (CNNs) were extended from artificial neural networks (ANNs) and customized for computer vision [7]. An example of a CNN is given in

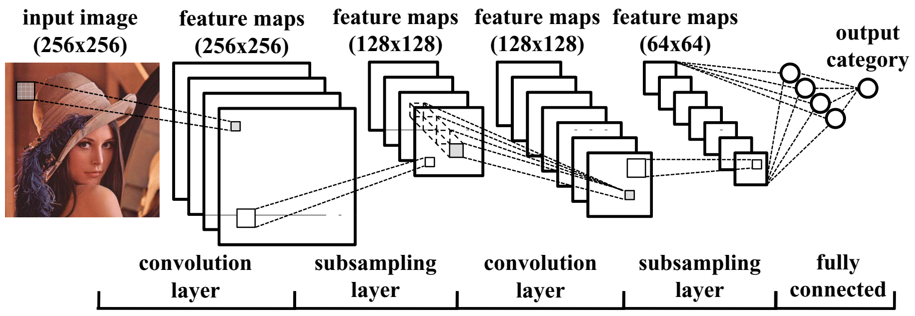

Fig. 1. An example of a convolutional neural network

Fig. 1. As shown in this figure, the intermediate results in a CNN are different sets of feature maps. The main working principle of a CNN is to gradually extract local features from feature maps of higher resolutions, and then to combine these features into more abstract feature maps of lower resolutions. This is realized by the two alternating types of layers in a CNN: convolution layers and subsampling layers. The last few layers in the CNN still use fully connected ANN classifiers to produce the abstracted classification results. The detailed computation patterns of different layers in the CNN are described as below:

Convolution Layer. In this layer, features, such as edges, corners, and crossings, are extracted from the input feature maps via different convolution kernels, and are combined into more abstract output feature maps. Assume there are Q input feature maps and R output feature maps, and the feature map size is $M \times N$. Also assume the convolution kernel size is $K \times L$. Then the computation in the convolution layer can be represented in a nested-loop description, as shown in Fig. 2. The array X contains the input feature maps, and the array Y contains the output feature maps which are initialized to zeros. The array W contains the weights in the convolution kernels. To regularize the computation pattern, we do not explicitly add the network bias to the output feature maps. Instead, we put a dummy input feature map of all 1's in array X and put the bias on the weights associated with this dummy input map in array W. The computational workload in the convolution layer is in the order of $O(R \cdot Q \cdot M \cdot N \cdot K \cdot L)$.

```
for(r=0; r<R; r++)             //output feature map
  for(q=0; q<Q; q++)           //input feature map
    for(m=0; m<M; m++)         //row in feature map
      for(n=0; n<N; n++)       //column in feature map
        for(k=0; k<K; k++)     //row in convolution kerenel
          for(l=0; l<L; l++)   //column in convolution kernel
            Y[r][m][n]+=W[r][q][k][l]*X[q][m+k][n+l];
```

Fig. 2. Example loop-nest representing the computation in a convolution layer of a CNN

Subsampling Layer. The purpose of this layer is to achieve spatial invariance by reducing the resolution of feature maps. In the example of Fig. 1, each feature map is scaled down by a subsample factor 2×2. The computational workload in this layer is in the order of $O(Q \cdot M \cdot N)$, which is much smaller than that in the convolution layer.

At the output of each layer, an activation function is further applied to each pixel in the feature maps to mimic the neuron activation.

2.2 Architecture of Real-Life CNN

The architecture of a real-life CNN that was used in the 2012 ImageNet contest [6] is shown in Fig. 3. It consists of eight layers. The first layer contains three 224×224 input images that are obtained from the original 256×256 image via data augmentation. The 1,000 neurons in the last layer report the likelihoods of the 1,000 categories that the input image might belong to. Layer 2 contains 96 feature maps, and each feature map is sized 55×55. They are partitioned into two sets, each containing 48 feature maps, so as to fit into two GPUs used in [6]. The other layers also follow notations similar to Fig. 3. Note that the convolution layer and the subsampling layer are merged together in Layers 1, 2, 3, and 6 of this architecture. There are no subsampling layers but only convolution layers in the other layers. The convolution kernel size is 11 in Layer 1, 5 in Layer 2, and 3 in the other layers. The default subsample factor is 2, except for a factor of 4 in Layer 1. The subsampling operations are not trainable in this architecture, but the function $\max$ is applied to the 2×2 or 4×4 pixel windows in each feature map, and is marked as max pooling in Fig. 3. The activation function is simplified to the Rectified Linear Unit (ReLU) function, $\max(0, x)$, as discussed in [6]. In Layer 2, 4, and 5, to avoid the inter-GPU communication, the features extracted from the two partitioned sets of input feature maps are not combined together at the output. The design choice of removing this combination is made by trial and error, and proves effective in [6].

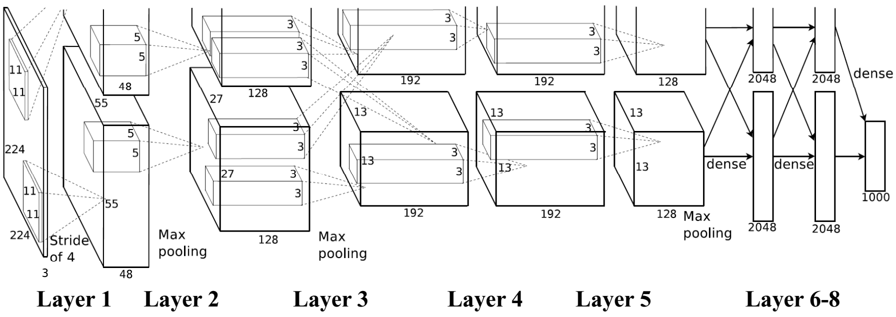

Fig. 3. A real-life CNN that was used in the 2012 ImageNet contest [6]

2.3 Runtime Breakdown of Real-Life CNN

To better understand the time-consuming part of the process of image recognition via a CNN, we reimplement the CNN in Fig. 3 in a single-thread CPU so that the workload can be measured by runtime. A breakdown of runtime is given in Table 1. We see that the runtime is dominated by the convolution layers. The main focus of this work is to optimize the computation in the convolution layers.

Table 1. Breakdown of CNN runtime in the recognition of 256 images

	Convolution Layer	Subsampling Layer	ReLU Activation	Fully Connected ANN
Layer 1	364s	720s	1.83s	-
Layer 2	1728s	416s	1.19s	-
Layer 3	5710s	147s	0.42s	-
Layer 4	2564s	-	0.42s	-
Layer 5	2652s	-	0.27s	-
Layer 6	-	29s	0.12	8.60s
Layer 7	-	-	0.12s	5.63s
Layer 8	-	-	0.03s	1.78s
Total	13018s	1313s	4.41s	16.0s
Breakdown	90.7%	9.15%	0.03%	0.11%

3 Properties of CNN Computation

3.1 Another View of CNN Computation

In this section we offer another view of the CNN computation in Fig. 2 that enables optimization of the computational workload. First denote the Q input feature maps as $x_1, x_2, ...x_Q$, and the R output feature maps as $y_1, y_2, ..., y_R$. Also denote the $R \times Q$ convolution kernels (each sized $K \times L$) as w_{rq} where $r = 1, 2, ..., R$ and $q = 1, 2, ..., Q$. We further denote the convolution operation between a kernel w_{rq} and a feature map x_q as

$$w_{rq} * x_q = z, \text{ where } z(m,n) = \sum_{k=0}^{K-1} \sum_{l=0}^{L-1} w_{rq}(k,l) x_q(m+k, n+l). \tag{1}$$

Here z represents the convolution result in the form of an $M \times N$ image, and $z(m,n)$ represents an image pixel in z. Then the computation in Fig. 2 can be represented as

$$\begin{array}{l} y_1 = w_{11} * x_1 + w_{12} * x_2 + \cdots + w_{1Q} * x_Q \\ y_2 = w_{21} * x_1 + w_{22} * x_2 + \cdots + w_{2Q} * x_Q \\ y_3 = w_{31} * x_1 + w_{32} * x_2 + \cdots + w_{3Q} * x_Q \\ \cdots \\ y_R = w_{R1} * x_1 + w_{R2} * x_2 + \cdots + w_{RQ} * x_Q \end{array} . \tag{2}$$

If we reorganize these x_q, y_r and w_{rq} in the form of column vectors and matrices

$$\overrightarrow{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_Q \end{pmatrix}, \overrightarrow{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_R \end{pmatrix}, W = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1Q} \\ w_{21} & w_{22} & \cdots & w_{2Q} \\ \vdots & \vdots & \ddots & \vdots \\ w_{R1} & w_{R2} & \cdots & w_{RQ} \end{pmatrix},$$

then the computation in Eq. (2) can be redefined as a special matrix/vector multiplication

$$\overrightarrow{y} = W \times \overrightarrow{x}.$$

Each element in the left operand W is a convolution kernel. Each element in the right operand $\overrightarrow{x}$ is an input feature map. Each element in the result $\overrightarrow{y}$ is an output feature map. The element-wise multiplication is redefined as the convolution between a kernel w_{rq} and a feature map x_q in Eq. (1). If a web server provider receives a batch of P images to recognize, we will have $\overrightarrow{x}_1, \overrightarrow{x}_2, \cdots, \overrightarrow{x}_P$ as parallel inputs, and $\overrightarrow{y}_1, \overrightarrow{y}_2, \cdots, \overrightarrow{y}_P$ as parallel outputs. Their computation can be merged as

$$(\overrightarrow{y}_1, \overrightarrow{y}_2, \cdots, \overrightarrow{y}_P) = (W\times\overrightarrow{x}_1, W\times\overrightarrow{x}_2, \cdots, W\times\overrightarrow{x}_P) = W\times(\overrightarrow{x}_1, \overrightarrow{x}_2, \cdots, \overrightarrow{x}_P).$$

This can be further simplified to a matrix multiplication

$$Y = W \times X,$$

where

$$X = (\overrightarrow{x}_1, \overrightarrow{x}_2, \cdots, \overrightarrow{x}_P),\ Y = (\overrightarrow{y}_1, \overrightarrow{y}_2, \cdots, \overrightarrow{y}_P).$$

Both the left operand X and the result Y are matrices of feature maps. This matrix multiplication representation provides a new view of the computation in convolution layers of CNNs. We name this representation Convolutional Matrix Multiplication (Convolutional MM).

3.2 Enabling New Optimization Opportunities

We can optimize the computation of Convolutional MM by revisiting techniques that have been built for normal matrix multiplication (Normal MM). For example, we can use the classical Strassen algorithm [11] to reduce the computational workload. In each recursion of matrix partitioning, the Strassen algorithm can reduce the number of multiplications by 1/8, but it incurs many extra additions. Note that in Normal MM, the element-wise multiplication is a multiplication between two numbers, while in our Convolutional MM, the element-wise multiplication is redefined as the convolution between a kernel and a feature map, which has a sufficiently high complexity to make the extra additions negligible. Our Convolutional MM is expected to experience more benefits from the Strassen algorithm than the Normal MM; this will be discussed in Section 4.

3.3 Properties of Convolutional MM

Before we go through any optimization, we first identify the properties of our Convolutional MM. If the addition of two convolution kernel matrices $W1$ and $W2$ of the same size are intuitively defined as the additions of all the pairs of weights at the same positions, i.e.,

$$W1 + W2 = W3,\ \text{where}\ w3_{rq}(k,l) = w1_{rq}(k,l) + w2_{rq}(k,l),$$

combined with the linearity of the operation defined in Eq. (1), we have

$$(W1 + W2) \times X = W1 \times X + W2 \times X. \tag{3}$$

Similarly, if the addition of two feature map matrices $X1$ and $X2$ of the same size are intuitively defined as additions of all the pairs of pixels at the same positions, we have

$$W \times (X1 + X2) = W \times X1 + W \times X2. \tag{4}$$

4 Computation Optimization

In this section we show how to extend the Strassen algorithm [11] from Normal MM to reduce the computational workload of our Convolutional MM. We start from

$$Y = W \times X,$$

where the number of elements in both rows and columns of Y, W, X is assumed to be even. We partition W, X and Y into equally sized block matrices

$$W = \begin{pmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{pmatrix}, \ X = \begin{pmatrix} X_{1,1} & X_{1,2} \\ X_{2,1} & X_{2,2} \end{pmatrix}, \ Y = \begin{pmatrix} Y_{1,1} & Y_{1,2} \\ Y_{2,1} & Y_{2,2} \end{pmatrix}.$$

Then we have

$$\begin{array}{l} Y_{1,1} = W_{1,1} \times X_{1,1} + W_{1,2} \times X_{2,1} \\ Y_{1,2} = W_{1,1} \times X_{1,2} + W_{1,2} \times X_{2,2} \\ Y_{2,1} = W_{2,1} \times X_{1,1} + W_{2,2} \times X_{2,1} \\ Y_{2,2} = W_{2,1} \times X_{1,2} + W_{2,2} \times X_{2,2} \end{array}. \tag{5}$$

Here, we still need 8 multiplications, the same number that we need in matrix multiplication before partitioning. We define new matrices

$$\begin{array}{l} M_1 := (W_{1,1} + W_{2,2}) \times (X_{1,1} + X_{2,2}) \\ M_2 := (W_{2,1} + W_{2,2}) \times X_{1,1} \\ M_3 := W_{1,1} \times (X_{1,2} - X_{2,2}) \\ M_4 := W_{2,2} \times (X_{2,1} - X_{1,1}) \\ M_5 := (W_{1,1} + W_{1,2}) \times X_{2,2} \\ M_6 := (W_{2,1} - W_{1,1}) \times (X_{1,1} + X_{1,2}) \\ M_7 := (W_{1,2} - W_{2,2}) \times (X_{2,1} + X_{2,2}) \end{array}. \tag{6}$$

Followed by the properties in Eq. (3) and Eq. (4), we can compute the result of the matrix multiplication from the 7 multiplications in Eq. (6) as follows:

$$\begin{aligned} Y_{1,1} &= M_1 + M_4 - M_5 + M_7 \\ Y_{1,2} &= M_3 + M_5 \\ Y_{2,1} &= M_2 + M_4 \\ Y_{2,2} &= M_1 - M_2 + M_3 + M_6. \end{aligned}$$

Here, we reduce the number of the redefined multiplications from 8 to 7 without changing the computation results. We can iterate this matrix partitioning process recursively until the submatrices degenerate into basic elements, i.e., separate convolution kernels and feature maps in our Convolutional MM. We can see that each recursion will reduce the number of multiplications by 1/8, but will incur 18 additions on the submatrices. In the Normal MM, all the elements are numbers, and either multiplications or additions are performed between these numbers. The overhead of 18 additions could completely eliminate the benefits brought by the multiplication savings in normal MMs. In our Convolutional MM, however, the element-wise multiplication is redefined as the convolution between a kernel and a feature map in Eq. (1). Suppose the convolution

Table 2. The FLOPS comparison of different operations in the Normal MM and our Convolutional MM

	element-wise addition	element-wise multiplication
the Normal MM	1	1
our Convolutional MM	$K \cdot L$ or $M \cdot N$	$2K \cdot L \cdot M \cdot N$

kernel size is $K \times L$, and the feature map size is $M \times N$. As shown in Table 2, the number of FLOPS (floating-point operations) in an element-wise multiplication will be $2K \cdot L \cdot M \cdot N$, which is much larger than either $K \cdot L$ FLOPS in a kernel addition or $M \cdot N$ FLOPS in a feature map addition. This makes the reduction of the number of multiplications very meaningful to our Convolutional MM.

5 Experimental Results

A comparison of our Convolutional MM with the Normal MM in terms of the savings of GFLOPS by the Strassen algorithm is shown in Fig. 4. We use the convolution kernel size 5×5 and the feature map size 55×55 in Layer 2 of Fig. 3, and sweep different square sizes for matrices W, X, Y in this experiment. We see that for the Normal MM, the Strassen algorithm may not bring benefits, but could lead to a $>100\%$ overhead, especially when the matrix size is small. In [4], even if the nested loops in Fig. 2 are unrolled and the computation is represented in a Normal MM to make the Strassen algorithm applicable, the Strassen algorithm still cannot bring too many benefits. But if the computation is represented in our Convolutional MM, much greater benefits can be achieved due to the redefined granularities of matrix elements and element-wise multiplications.

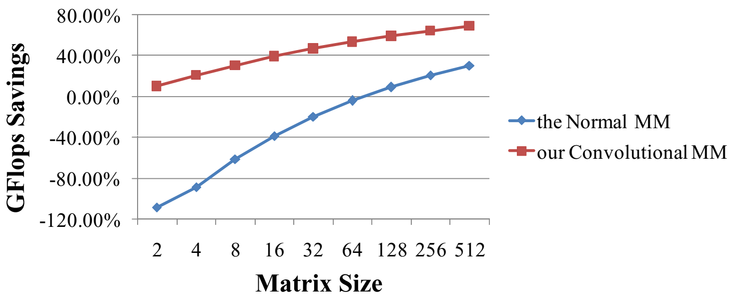

Fig. 4. Comparison of our Convolutional MM with the Normal MM in terms of the savings of GFLOPS by the Strassen algorithm

A sensitivity study on the convolution kernel size and the feature map size is provided in Fig. 5. Here we fix the matrix size to 256. As shown in Fig. 5(a), the convolution kernel size has a high impact on GFLOPS savings. This matches the analysis that the

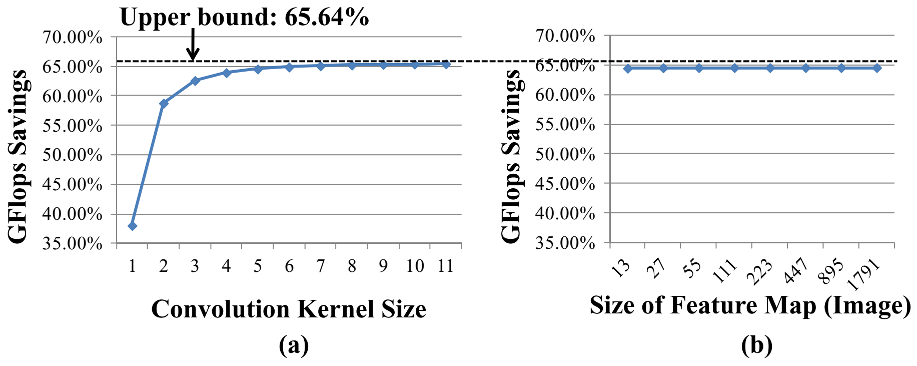

Fig. 5. A sensitivity study on convolution kernel size and feature map size

FLOPS difference between the element-wise multiplication and the element-wise addition of our Convolutional MM is proportional to the kernel size (assume the feature map size is much larger than the kernel size). Since the matrix size is limited to 256, there could be at most 8 recursions of the Strassen algorithm, which imposes an upper bound of $1 - (7/8)^8 = 65.64\%$ on the GFLOPS savings. Fig. 5(a) shows that we approach this upper bound as the convolution kernel size increases. The GFLOPS savings are invariant with the increase of the feature map size, as shown in Fig. 5(b), since the computational workloads of both the element-wise multiplication and addition will increase.

We reimplement the real-life CNN in Fig. 3 and apply the Strassen algorithm to reduce the computational workload. Experimental results are listed in Table 3. As shown in this table, no matrices are square, and no matrices have sizes equal to the power of 2. To deal with the non-square matrices, we stop matrix partitioning once either the row size or the column size becomes small. To solve the not-the-power-of-2 problem, we pad a dummy row or column in the matrices once we encounter an odd number of rows or columns during matrix partitioning. Note that the Strassen algorithm is based on recursive matrix partitioning, which is a cache-oblivious algorithm that can take advantage of a CPU cache without knowing the cache size. For the sake of

Table 3. Workload reduction by extending the Strassen algorithm to the real-life CNN in Fig. 3 via our Convolutional MM

		Layer 1	Layer 2	Layer 3	Layer 4	Layer 5
Matrix Parameters	Q	3	48	256	192	192
	R	96	128	384	192	128
	K, L	11	5	3	3	3
	M, N	224	55	27	13	13
Runtime	original	364s	865s	5710s	1282s	1326s
	our optimization	433s	864s	3863s	683s	998s
	savings	-18%	27%	32%	47%	24%

fairness, we also implement the baseline matrix multiplication in a cache-oblivious algorithm in Eq. (5). The hardware platform is a Xeon server with CPUs running at 2GHz. We limit the number of threads initialized by our Convolutional MM computation to 1 since we are measuring the reduction of total workloads by the runtime. Table 3 shows that we can get up to a 47% savings in certain convolution layers. Note that this gain is achieved without any change in image recognition results or the addition of any hardware accelerators.

6 Conclusion

In this work the computation in the convolution layers of a CNN is expressed in a new representation — Convolutional Matrix Multiplication (Convolutional MM). This representation helps identify the linear algebraic properties of the CNN computation, and enables extension of state-of-art algorithms that have been built for Normal Matrix Multiplication (Normal MM) to CNNs for computational workload reduction. This kind reduction does not change any image recognition results, and does not require any extra hardware accelerators. We use the Strassen algorithm as an example to show the necessary algorithmic extension from Normal MM to our Convolutional MM, and to show the extra benefits that can be gained by this Convolutional MM. Our methodology is verified on a real-life CNN. Experimental results show that we can reduce the computation by up to 47%. More well-studied algorithms on linear algebra can be further extended to our Convolutional MM to optimize the CNN computation.

Acknowledgements. This research is supported by the NSF Expedition in Computing Award CCF-0926127 and by C-FAR (one of six centers of STARnet, an SRC program sponsored by MARCO and DARPA).

References

1. ImageNet Contest (2012), `http://www.image-net.org/challenges/LSVRC/2012/index`
2. Behnke, S.: Hierarchical Neural Networks for Image Interpretation. LNCS, vol. 2766. Springer, Heidelberg (2003)
3. Chakradhar, S., Sankaradas, M., Jakkula, V., Cadambi, S.: A dynamically configurable co-processor for convolutional neural networks. In: International Symposium on Computer Architecture, p. 247 (2010)
4. Chellapilla, K., Puri, S., Simard, P.: High Performance Convolutional Neural Networks for Document Processing. In: International Workshop on Frontiers in Handwriting Recognition (2006)
5. Farabet, C., Poulet, C., Han, J.Y., LeCun, Y.: CNP: An FPGA-based processor for Convolutional Networks. In: International Conference on Field Programmable Logic and Applications, vol. 1, pp. 32–37 (August 2009)
6. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet Classification with Deep Convolutional Neural Networks. In: Proceedings of Neural Information and Processing Systems, pp. 1–9 (2012)

7. Lecun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE 86, 2278–2324 (1998)
8. Osadchy, M., Yann, L.C., Matthew, L.M.: Synergistic Face Detection and Pose Estimation with Energy-Based Models. Journal of Machine Learning Research 8, 1197–1215 (2007)
9. Peemen, M., Setio, A.A.A., Mesman, B., Corporaal, H.: Memory-centric accelerator design for Convolutional Neural Networks. In: International Conference on Computer Design, pp. 13–19 (October 2013)
10. Simard, P., Steinkraus, D., Platt, J.: Best practices for convolutional neural networks applied to visual document analysis. In: International Conference on Document Analysis and Recognition (ICDAR), vol. 1, pp. 958–963 (2003)
11. Strassen, V.: Gaussian elimination is not optimal. Numerische Mathematik 13(4), 354–356 (1969)