J Supercomput (2019) 75:1065–1077
https://doi.org/10.1007/s11227-018-2300-2

Caffe CNN-based classification of hyperspectral images on GPU

Alberto S. Garea[1] · **Dora B. Heras**[1] · **Francisco Argüello**[2]

Published online: 9 March 2018

Abstract Deep learning techniques based on Convolutional Neural Networks (CNNs) are extensively used for the classification of hyperspectral images. These techniques present high computational cost. In this paper, a GPU (Graphics Processing Unit) implementation of a spatial-spectral supervised classification scheme based on CNNs and applied to remote sensing datasets is presented. In particular, two deep learning libraries, Caffe and CuDNN, are used and compared. In order to achieve an efficient GPU projection, different techniques and optimizations have been applied. The implemented scheme comprises Principal Component Analysis (PCA) to extract the main features, a patch extraction around each pixel to take the spatial information into account, one convolutional layer for processing the spectral information, and fully connected layers to perform the classification. To improve the initial GPU implementation accuracy, a second convolutional layer has been added. High speedups are obtained together with competitive classification accuracies.

Keywords Hyperspectral · Classification · Convolutional neural network · Deep learning · Caffe · GPU · CuDNN

✉ Alberto S. Garea
jorge.suarez.garea@usc.es

Dora B. Heras
dora.blanco@usc.es

Francisco Argüello
francisco.arguello@usc.es

[1] Centro singular de Investigación en Tecnoloxías da Información (CiTIUS), Universidade de Santiago de Compostela, Santiago de Compostela, Spain

[2] Departamento de Electrónica y Computación, Universidade de Santiago de Compostela, Santiago de Compostela, Spain

1 Introduction

Hyperspectral images contain a large amount of information that can be exploited during the processing. This information is not only spectral but there is also a lot of spatial information to be considered in the neighborhood of each pixel. Hyperspectral techniques that can exploit both types of information are known as spectral-spatial techniques [1]. When these techniques are introduced in the classification of hyperspectral images, experimental results show high accuracy improvements.

Recently, deep learning techniques have been introduced in the field of classification of hyperspectral datasets [2–7]. Applications include pattern recognition and statistical classification. These classifiers consist of several layers with nonlinear processing units to extract and transform different features. Each layer uses the output of the previous layer as input and the network can be trained in a supervised or unsupervised manner. The proposed methods extract spatial information using structures such as Multilayer Perceptrons (MLP) or Convolutional Neural Networks (CNNs). Usually before the extraction of spatial information, a dimensionality reduction is performed using techniques such as Principal Component Analysis (PCA), Independent Component Analysis (ICA) or wavelets in order to obtain moderately small vectors.

A CNN contains convolutional layers that can be used to perform spatial convolutions on the hyperspectral image bands. Usually pooling layers are also included in order to apply some kind of decimation and reduce the number of coefficients. A CNN may have one or more convolutional layers [2–7], but the final classification is performed using one or more fully connected layers. Activation functions to introduce nonlinearity, usually of sigmoid type, can be included in convolutional layers. Such functions are similar to those used in MLPs. Usually the backpropagation algorithm is used to set the coefficients of both, neurons of fully connected layers and convolution filters.

Some published deep learning schemes applied to hyperspectral images use only the spectral information. Thus, Hu et al. [3] propose a scheme which does not consider spatial information since each input is a single pixel-vector. Other schemes incorporate the spectral and spatial information separately to the classifier, often constructing a stack-vector for input to the neural network and using PCA [2,4–6].

Remote sensing hyperspectral applications are computationally demanding and, therefore, good candidates to be projected in high performance computing infrastructures such as clusters or specialized hardware devices [8]. GPUs provide a cost-efficient solution to carry out onboard real-time processing of remote sensing hyperspectral data for performing hyperspectral unmixing, classification or change detection, among others [9]. In the case of deep learning techniques, different high-level frameworks optimized for GPU computing are available, such as Theano, Caffe [10], TensorFlow or Torch. Specific GPU libraries of primitives for deep neural networks such as cuDNN [11] can be used to perform common operations in CNNs, for example, forward and backward convolution or pooling. The implementations of deep learning methods for hyperspectral images are in some cases presented in terms of execution times but without an analysis of the computational cost [2,6]. In other cases the use of an optimized framework such as Caffe [12] is mentioned but without including execution times or a detailed analysis of the implementations.

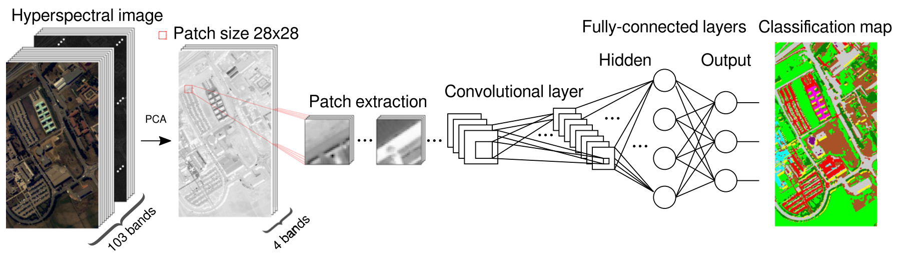

Fig. 1 HYCNN scheme for the classification of hyperspectral images considering 28×28 patches

In this paper we propose a CUDA GPU spectral-spatial classification scheme for hyperspectral images based on CNNs. A comparative study in the Caffe framework between the Caffe functions and the cuDNN library functions to reduce the execution time has been performed. Different configurations for both implementations are analyzed. The effect of including more than one convolution is also studied.

The paper is organized as follows: Sect. 2 presents the proposed spectral-spatial classification scheme in CPU, Sect. 3 presents the GPU code. The evaluation is performed in Sect. 4, and, finally, Sect. 5 presents the conclusions.

2 Spectral-spatial CNN-based classification

In this section we present an approach for the classification of hyperspectral images based on PCA, patch extraction, and CNNs that we called HYCNN. Figure 1 shows the operations performed and the network structure. These are described in more detail in the pseudocode of Algorithm 1.

Algorithm 1 Steps of the HYCNN scheme

```
Input: Hyperspectral image
Output: Classification map
Parameters:
$M$: number of pixels
$N_0$: number of bands
$N_1$: principal components
$H \times V$: patch size
$E$: number of epochs
$I$: number of iterations for each epoch
$B$: batch size
$N_2$: number of convolution filters
$F_1 \times F_2$: spatial size of filters
$D_1 \times D_2$: decimation factor
$N_3$: number of neurons in hidden layer
$N_4$: number of neurons in the output layer
$\eta$: learning parameter

1. Preprocessing
1.1 PCA on the image
2. Patch extraction
2.1 Patch around each pixel
3. Convolutional layer
3.1 Convolution filtering
3.2 Pooling (average)
3.3 Activation function (sigmoid)
4. Fully connected layers
4.1 Hidden layer (with sigmoid)
4.2 Output layer (with sigmoid)
```

As a first step, HYCNN performs a dimensionality reduction using PCA. Then, a patch is extracted around each pixel to be classified. This step aims at considering the spatial information in the neighborhood of a pixel in addition to the spectral information. Accordingly, the patch has the same number of components as those retained from the PCA, and comprises a window around the pixel. The window size is an adjustable parameter of the classification. Each patch is considered a sample and used as the unit of information during the training and classification phases by the CNN.

The next step is the processing of each patch by the CNN. This consists of: convolutional filters, pooling layer and activation function. A convolutional layer is a locally connected structure which is convolved with the image to produce several feature maps, one for each filter. Each filter consists of a rectangular grid of neurons. Unlike a fully connected layer, the filter coefficients used in all the nodes are the same.

The convolutional layer of our scheme processes several components (spectral bands). The inputs to the filters are the patches, which we assume to have in this sequential algorithm a size of $H \times V \times N_1$, being H and V the size of the spatial dimensions, and N_1 the number of reduced features in the spectral dimension. In order to extract multiple features, the convolutional layer comprises N_2 filters, so we have the same number of 2D maps at the output. Regarding the size of the filters, if $F_1 \times F_2$ is the size of the spatial grid, each filter has $F_1 \times F_2 \times N_1$ coefficients. Multiple consecutive convolution layers can be applied to increase the accuracy/quality of the results.

The pooling layer takes small rectangular blocks from the convolutional layer and subsamples them to produce a single output from each block. For the pooling layers each map is subsampled with mean pooling over blocks of size $D_1 \times D_2$. After the subsampling, a sigmoidal nonlinearity is applied to each feature map.

The last part of the scheme consists of fully connected layers, which perform the high-level reasoning of the CNN. A fully connected layer takes all the outputs in the previous layer and connects then to every single neuron it has. This type of layer is arranged in one dimension, so there are no spatiality in the operations anymore. In this paper we use the typical two-layer MLP, with hidden and output layers. The number of neurons in the hidden layer is the adjustable parameter N_3, while the output layer has a number of neurons, N_4, equal to the number of classes in the hyperspectral image. The activation function in both, convolutional and fully connected layers, is of sigmoid type.

The learning of all the layers of the CNN in this scheme is conducted using a backpropagation algorithm. The error is computed at the output of the network using the training samples and comparing the results to the reference data. Then, the error is propagated backwards through the network. The backpropagation is used in conjunction with an optimization method, in this case a gradient descent. It calculates the gradient of a cost function with respect to all the weights of the network, and then updates the weights in an attempt to minimize the cost function. The learning parameter, usually denoted as η, indicates how much the weights are adjusted at each update.

The computational complexity of the proposed scheme (HYCNN) can be summarized as the sum of the cost of steps 1, 3 and 4 in Algorithm 1. For step 1, PCA calculation, the computational complexity is $O({N_0}^2 M + {N_0}^3)$ where N_0 is the number of bands of the hyperspectral input image, and M is the number of pixels of the hyperspectral image. In the case of step 3, the convolutional layer, the cost is $O(BHVN_1F_1F_2N_2)$ where B is the batch size, i.e., the number of patches used at the same time in each iteration, $H \times V$ is the size of the patch, N_1 is the number of bands of the patch, $F_1 \times F_2$ is the size of the filter, and N_2 is the number of filters. The last step with relevant cost is step 4, fully connected layer. The computational cost of this layer is $O(N_2 P_1 P_2 N_3 N_4)$ where N_2 is the number of filters, P_1 is the quotient between H and the decimation factor D_1 applied by the pooling layer, P_2 is the quotient between V and the other decimation factor D_2 applied by the pooling layer, N_3 is the number of neurons in the hidden layer, and N_4 refers to the number of classes in the hyperspectral image. Therefore the computational cost is $O({N_0}^2 M + {N_0}^3 + EI(BHVN_1F_1F_2N_2 + N_2P_1P_2N_3N_4))$, being E is the number of times (epochs) that the set of samples applies to the network, and I is the number of iterations for each epoch.

3 Spectral-spatial CNN-based classification in GPU

In this section we introduce some Compute Unified Device Architecture (CUDA) programming fundamentals as well as the CUDA GPU implementation of the scheme proposed in Sect. 2.

3.1 CUDA GPU programming fundamentals

CUDA is a parallel computing platform and programming model that enables NVIDIA GPUs to execute programs invoking parallel functions called kernels [13]. Each kernel launches a user-defined number of threads that are organized into blocks. The blocks are arranged in a grid that is mapped to a hierarchy of CUDA cores in the GPU. Threads can access data from multiple memory spaces. Each block has a shared memory that is visible exclusively to the threads within this block and whose lifetime is equal to the block lifetime. The shared memory lifetime makes it difficult to share data among thread blocks. This implies the use of global memory whose access is slower than shared memory access. The new Pascal architecture has introduced changes regarding the memory hierarchy such as an increase in the size of the memory up to 96 Kb, besides it incorporates GDDR5X type memory that supports 10 Gbit/s [14].

Different performance optimization strategies have been applied in this work. The most important one is to reduce the data transfers between the CPU and the GPU memories. Another key aspect is to improve the efficiency in the use of the memory hierarchy by performing the maximum number of computations on the data already stored in shared memory. The search for the best kernel configurations is also fundamental. To obtain the highest possible occupancy is the only way to hide latencies and keep the hardware busy. To achieve this, the maximum block size for each kernel is selected with the requirement that the number of registers and the shared memory usage do not act as occupancy limiters. Finally, the existing CUDA optimized libraries must be used. CULA [15], MAGMA [16], and CUBLAS [17] are employed for algebra operations. For the deep learning calculations the Caffe framework is used. Furthermore, the cuDNN library is used to perform several deep neural network operations and to compare to the Caffe implementation of the same operations.

3.2 CUDA implementation

In this section the GPU implementation of the HYCNN algorithm described in Sect. 2 is detailed. The pseudocode in Algorithm 2 shows a detailed description of the classification scheme using Caffe with calls to the cuDNN library (cuDNN-HYCNN). In the case of the implementation in Caffe without cuDNN calls (Caffe-HYCNN) the operations are performed by functions of the Caffe framework. Our interest is in the comparison of both implementation options as we will show in Sect. 4. The kernels executed in GPU are placed between $<>$ symbols. The pseudocode also includes the GM and SM acronyms to indicate kernels executed only in global memory and kernels that use shared memory, respectively. The whole forward–backward process for the training phase of the algorithm is detailed. The CNN is implemented using Caffe. Since the calls to Caffe functions produce a high number of calls to libraries, these are grouped in the pseudocode by steps of the scheme and only the most repeated kernels are included pointing out the call sequence. In this pseudocode all the possible calls to cuDNN are performed.

As a first step, the PCA algorithm using EVD (EVD-PCA) is applied to reduce the dimensionality of the dataset. For details of the GPU implementation see [18]. A patch is then extracted around each pixel and stored in two different Lightning Memory-Mapped Databases (LMDBs) to be accessed from the Caffe framework. The first database stores the training patches, whereas the second one stores the test patches. Both, the CNN steps and the fully connected layers steps are applied to each patch N times (epochs).

The training phase is divided into two main steps: forward and backward. The forward step computes all the training patches through the full network to obtain a classification result and the backward step updates the network weights to adjust the obtained classification result.

The forward step starts by applying the convolution filters to each training patch. Unlike in the CPU implementation where the $H \times V$ pixels of the patch are computed through the convolution filters sequentially, in the GPU implementation all patches are processed in parallel using the cuDNN function called cuDNNConvolutionForward (line 3 in the pseudocode 2). Next, the biases are added (line 4). Finally, the kernel sync_conv_gropus (line 5) computes a synchronization operation over the results from previous kernels.

Next, a pooling substep is performed using a cuDNN kernel called pooling_fw_4d (line 6). This function computes the pooling over all the training patches at the same time. The last substep of the CNN is the activation. The cuDNNSigmoid-Layer::Forward_gpu() calls the cuDNN function to perform the sigmoid activation (line 7).

Once the CNN has finished, two fully connected layers perform the classification. First, an inner product function (line 8) that uses CuBLAS multiplies the CNN output matrix by a matrix of learned weights. Next, a sigmoid activation function (line 9) is applied over the previous result using a cuDNN function. The previous two operations are repeated over the last fully connected layer (lines 10 and 11).

At this point, the output of the full network contains the classification of each training patch. Then, a softmax function (line 12) is applied to obtain a probability

Algorithm 2 HYCNN classifier for hyperspectral images using cuDNN calls in the Caffe framework (cuDNN-HYCNN)

```
    Input: Hyperspectral image

    GPU EVD-PCA algorithm

 1: for each iteration do
    Forward
    Convolution filtering
 2:     cuDNNConvolutionLayer::Forward_gpu():
 3:         cuDNNConvolutionForward() → <computeOffsetsKernel>, <scuDNN_128x32>                 ▷ SM + GM
 4:         cuDNNAddTensor() → <add_tensor>                                                      ▷ GM
 5:         <sync_conv_groups>                                                                   ▷ GM
    Average Pooling
 6:     cuDNNPoolingLayer::Forward_gpu() → cuDNNPoolingForward() → <pooling_fw_4d>               ▷ SM + GM
    Convolution Activation
 7:     cuDNNSigmoidLayer::Forward_gpu() → cuDNNActivationForward() → <activation_fw_4d>         ▷ GM
    First Inner
 8:     InnerProductLayer::Forward_gpu() → caffe_gpu_gemm() → <sgemm_128x64>, <gemmk1>           ▷ SM + GM
    First Inner activation
 9:     cuDNNSigmoidLayer::Forward_gpu() → cuDNNActivationForward() → <activation_fw_4d>         ▷ GM
    Second Inner
10:     InnerProductLayer::Forward_gpu() → caffe_gpu_gemm() → <sgemm>, <gemmk1>                  ▷ SM + GM
    Second Inner Activation
11:     cuDNNSigmoidLayer::Forward_gpu() → cuDNNActivationForward() → <activation_fw_4d>         ▷ GM
    SoftMax with Loss
12:     cuDNNSoftmaxLayer::Forward_gpu() → cuDNNSoftmaxForward() → <softmax_fw>                  ▷ SM + GM
13:     SoftmaxLossForwardGPU() → <SoftmaxLossForwardGPU>, <cublasSasum>                         ▷ GM

    Backward
    Second Inner Activation
14:     cuDNNSigmoidLayer::Backward_gpu() → cuDNNActivationBackward() → <activation_bw_4d>       ▷ GM
    Second Inner
15:     InnerProductLayer::Backward_gpu() → <sgem_largeK>, <gemv2N>, <sgemm_128x64>              ▷ SM + GM
    First Inner Activation
16:     cuDNNSigmoidLayer::Backward_gpu() → cuDNNActivationBackward() → <activation_bw_4d>       ▷ GM
    First Inner
17:     InnerProductLayer::Backward_gpu() → <sgemm_128x64>, <gemv2N>, <sgemm_128x64>             ▷ SM + GM
    Convolution Activation
18:     cuDNNSigmoidLayer::Backward_gpu() → cuDNNActivationBackward() → <activation_bw_4d>       ▷ GM
    Pooling
19:     cuDNNPoolingLayer::Backward_gpu() → cuDNNPoolingBackward() → <pooling_bw_kernel_avg>     ▷ GM
    Convolution filtering
20:     cuDNNConvolutionLayer::Backward_gpu():
21:         cuDNNConvolutionBackwardBias() → <calc_bias_diff>                                    ▷ SM + GM
22:         cuDNNConvolutionBackwardFilter() → <copmuteWgradOffsets>                             ▷ GM
23:         cuDNNConvolutionBackwardData() → <copmuteBOffsets>, <scuDNN_128x32_strideB>          ▷ SM + GM
24:         <sync_conv_groups>                                                                   ▷ GM
    Weights update
25:     caffe::SGDSolver() → <SGDUpdate>                                                         ▷ GM
26: end for
```

distribution over classes. This function takes a vector of arbitrary real-valued scores and converts it to a vector of values between zero and one that sum one. The last substep of the forward pass is to compute the loss of the network using the SoftmaxLossForwardGPU function (line 13).

Regarding the backward step, it includes all the substeps of the forward step but applied in reverse order (lines 14–24). This allows to update the values of all the neurons in the full network based on the results of the loss function computed in the forward step. At the end of the loop, the update of all the weights of the full network is performed (line 25).

4 Results

This section shows the experimental results obtained for the two GPU implementations of the HYCNN scheme (using the Caffe framework with and without calls to cuDNN) comparing to the CPU one. This comparison is carried out in terms of computation time and classification accuracy.

The proposed algorithms have been evaluated on a PC with a quad-core Intel i5-6600 at 3.3GHz and 32 GB of RAM. The codes have been compiled using the gcc 4.8.4 version with OpenMP (OMP) 3.0 support under Linux using four threads. The OPENBLAS library has been used to accelerate the algebra operations included in the algorithms. Regarding the GPU implementation, CUDA codes run on an Pascal NVIDIA GeForce GTX 1070 with 15 Streaming Multiprocessors (SMs) and 128 CUDA cores each. The CUDA codes have been compiled under Linux using the nvcc version 8.0.61 of the toolkit. As usual in remote sensing [19], measures of classification accuracy are given in terms of overall accuracy (OA), which is the percentage of correctly classified pixels comparing to the reference data information available. The computational performance results are expressed in terms of execution times and speedups. The results are the average of 10 independent executions.

Four remote sensing datasets are considered: a 103-band ROSIS image of the University of Pavia (Pavia U.), a 220-band AVIRIS image taken over Northwest Indiana (Indiana), a 204-band AVIRIS image taken over Salinas Valley, California (Salinas) and a 102-band ROSIS image taken over Pavia, northern Italy (Pavia C.). The images and the corresponding reference data are shown in Fig. 2.

For each dataset the samples of the reference data are randomly distributed between the training [18] and testing sets. During the testing stage all pixels of the image are classified, but the samples used in the training stage are excluded for the calculation of the accuracy results (see Table 1).

The configuration parameters were determined by performing experiments varying the number of principal components, the batch size and the filter size for the code executed in CPU, and also for both GPU implementations. It is important to point out that for each one of the implementations in Table 2 the parameters that achieve the best accuracy values (OA) were used. The parameters that are common to all images are $H = V = 28$ (patch size), $N_1 = 4$ (number of principal components), $F_1 = F_2 = 5$ (filter size), and $D_1 = D_2 = 2$ (decimation factor). For the remaining parameters the values in CPU are $N_2 = 16$ (number of filters) and $N_3 = 100$ (neurons in the hidden layer). The values for the GPU implementations are $N_2 = 16$ for Pavia C. and Indiana, $N_2 = 24$ for Salinas and Pavia U., $N_3 = 100$ for Pavia C. and Indiana, $N_3 = 200$ for Salinas, and, finally, $N_3 = 300$ for Pavia U. The backpropagation algorithm was run in CPU with learning parameter $\eta = 0.2$ and in GPU with $\eta = 0.2$ for Pavia U. and Pavia C., $\eta = 0.01$ for Salinas, and $\eta = 0.4$ for Indiana. The batch size in the CPU implementation is equal to the number of training samples and in the GPU implementation it was reduced to 256 for all the images except for Salinas, that uses a batch size equal to 128. For both GPU implementations, Caffe-HYCNN and cuDNN-HYCNN, a block size of 512 was used. The value was decided after analyzing the execution time for the different block sizes detailed in Table 3. The 512-thread block size offers reasonable good results for all the images.

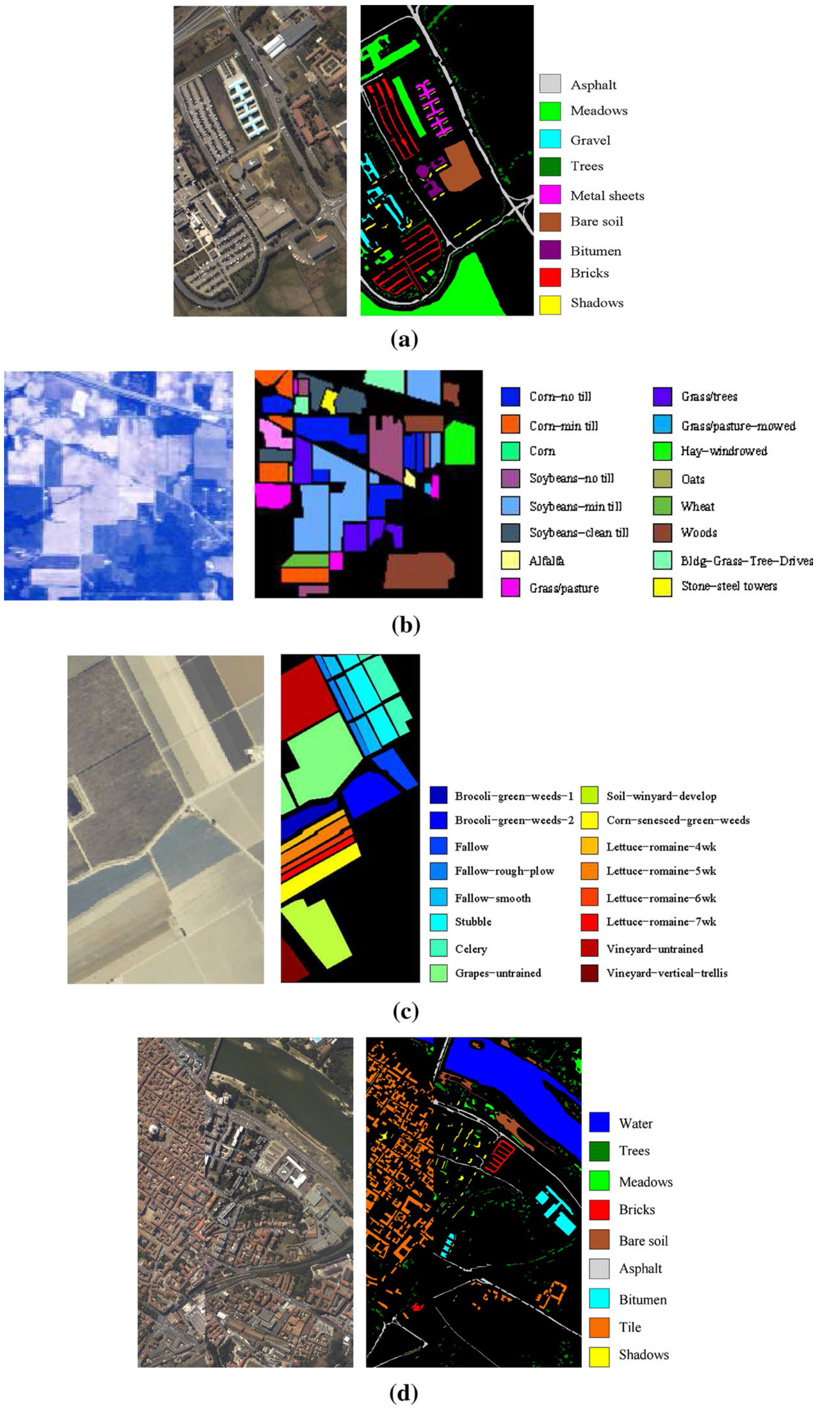

Fig. 2 Hyperspectral datasets and reference data: (a) *Pavia U.*, (b) *Indiana*, (c) *Salinas*, (d) *Pavia C.*

Table 1 Information for the test remote sensing datasets

Datasets	Sensor	Classes	Dimensions	Samples	Training samples (%)
Pavia U.	ROSIS	9	610×340×103	42776	3921 (9.17)
Indiana	AVIRIS	16	145×145×220	10249	695 (6.78)
Salinas	AVIRIS	16	512×217×204	53053	1076 (2.02)
Pavia C.	ROSIS	9	1096×715×102	140696	7456 (5.30)

Table 2 Execution times (training and test steps), speedups and classification accuracies. Comparative between the CPU and GPU (Caffe-HYCNN and cuDNN-HYCNN) implementations of the HYCNN scheme

	CPU		Caffe-HYCNN			cuDNN-HYCNN		
Dataset	Time (s)	OA (%)	Time (s)	OA (%)	Speedup	Time (s)	OA (%)	Speedup
Pavia U.	1404.26	98.50	74.14	97.20	18.94×	18.92	97.17	74.22×
Indiana	244.97	84.45	40.38	84.02	6.07×	14.14	82.81	17.32×
Salinas	400.10	97.17	76.35	89.41	5.24×	19.28	89.35	20.75×
Pavia C.	2631.63	99.41	104.23	97.99	25.25×	36.73	97.99	71.65×

Table 3 Block size comparative for the cuDNN-HYCNN implementation using only one convolutional layer

	Pavia U.		Indiana		Salinas		Pavia C.	
Block size	Time	OA (%)	Time	OA (%)	Time	OA (%)	Time	OA (%)
64	18.52s	96.48	14.03s	82.45	19.33s	89.82	36.74s	98.03
128	18.51s	95.91	14.19s	82.80	19.30s	89.58	36.93s	97.96
256	18.53s	92.52	14.03s	83.03	19.31s	87.35	36.71s	97.98
512	18.92s	97.18	14.14s	82.81	19.28s	89.35	36.73s	97.99
768	18.51s	91.83	14.04s	83.76	19.32s	89.43	36.71s	98.02

Table 2 shows the execution times and speedups for the whole classification scheme for the four datasets (including the training and testing steps and also the PCA step) as well as the classification accuracies for the CPU and the two GPU implementations (Caffe-HYCNN and cuDNN-HYCNN). The differences in classification accuracy among the CPU and the GPU schemes are produced mainly by the weights update during the backpropagation. For the CPU case the update is carried out for each sample separately, on the contrary for the GPU case the updates are performed by blocks of samples (batch). As a consequence the number of epochs required by the CPU and the GPU implementations is different.

The reduction of time obtained by cuDNN-HYCNN over Caffe-HYCNN may be due to the patch that is transformed into an intermediate structure in the Caffe-HYCNN version. In particular, for the convolution in the forward step, the speedups achieved by the cuDNN-HYCNN and the Caffe-HYCNN implementations are of 195.77× and 17.91×, respectively, for Pavia U. as shown in Table 4. In the case of Salinas, the

Table 4 Execution times and speedups for the training step of the HYCNN scheme for the Pavia U. dataset comparing the CPU and the GPU implementations (Caffe-HYCNN and cuDNN-HYCNN)

Step	Lines	CPU	Caffe-HYCNN	Speedup	cuDNN-HYCNN	Speedup
Forward step						
Convolution	2–5	413.07s	23.06s	17.91×	2.11s	195.77×
Average Pooling	6	7.11s	0.47s	14.97×	0.51s	13.94×
Convolution Act.	7	52.33s	0.41s	127.98×	0.19s	275.42×
First Inner	8	384.63s	1.68s	229.34×	2.09s	184.03×
First Inner Act.	9	2.56s	0.04s	62.72×	0.02s	128×
Second Inner	10	1.24s	0.07s	17.90×	0.07s	17.71×
Second Inner Act.	11	0.25s	0.01s	31.95×	0.01s	25×
Loss	12–13	0.05s	0.13s	0.39×	0.12s	0.42×
Backward step						
Second Inner	14–15	0.70s	0.20s	3.42×	0.21s	3.33×
First Inner	16–17	190.96s	1.35s	141.84×	1.36s	140.41×
Convolution	18–24	322.62s	44.09s	7.32×	7.04s	45.83×
Total		1375.52s	71.51s	19.24×	13.73s	100.11×

Table 5 Execution times and speedups for the training step of the HYCNN scheme for the Salinas dataset comparing the CPU and the GPU implementations (Caffe-HYCNN and cuDNN-HYCNN)

Step	Lines	CPU	Caffe-HYCNN	Speedup	cuDNN-HYCNN	Speedup
Forward step						
Convolution	2-5	111.74s	21.81s	5.12×	2.13s	52.46×
Average Pooling	6	1.94s	0.50s	3.88×	0.53s	3.66×
Convolution Act.	7	14.14s	0.15s	95.90×	0.14s	101×
First Inner	8	103.75s	1.38s	74.93×	1.43s	72.55×
First Inner Act.	9	0.69s	0.04s	17.09×	0.03s	23×
Second Inner	10	0.58s	0.09s	6.16×	0.10s	5.8×
Second Inner Act.	11	0.11s	0.02s	4.72×	0.01s	11×
Loss	12–13	0.01s	0.25s	0.05×	0.20s	0.05×
Ba-ckward step						
Second Inner	14–15	0.33s	0.37s	0.89×	0.25s	1.32×
First Inner	16–17	53.25s	1.14s	46.86×	1.17s	45.51×
Convolution	18–24	86.96s	41.53s	2.09×	6.98s	12.46×
Total		373.50s	67.28s	5.55×	12.97s	28.82×

speedups for the forward step of the convolution shown in Table 5 are also much higher for cuDNN-HYCNN. The results in both tables correspond to the aggregate values for all the training iterations and show that the convolution is the most time consuming function.

Table 6 Execution times, classification accuracies and number of epochs for the cuDNN-HYCNN implementation with one and two convolutional layers

	cuDNN-HYCNN (1 CNN)			cuDNN-HYCNN (2 CNN)		
Dataset	Time (s)	OA (%)	# Epochs	Time (s)	OA (%)	# Epochs
Pavia U.	18.92	97.17	326	22.63	97.58	261
Indiana	14.14	82.81	1473	7.61	89.11	737
Salinas	19.28	89.35	1190	24.28	90.81	952
Pavia C.	36.73	97.99	343	45.13	98.15	275

To improve the accuracy of the HYCNN scheme another convolutional layer has been added just after the first one. Table 6 shows the execution times for all the datasets (including the training and testing steps and also the PCA step) as well as the classification accuracies and the number of epochs required. Different setups have been used for the different datasets. In particular, in the cuDNN-HYCNN implementation with two convolutions, the output size for the first convolution is $32\times12\times12$ (number of filters $\times$ width $\times$ height of the filter) for all the datasets except for Indiana, that uses $16\times$ 12×12. In the same way, the output for the second convolution is $64\times4\times4$ for all the datasets except Indiana, that uses $16\times4\times4$. Note that the cuDNN-HYCNN implementation with two convolutions increases the accuracy of the classification for all the images at the cost of a slight increase in the execution times.

5 Conclusions

In this paper we analyze the efficiency of a proposed GPU classification scheme for hyperspectral images based on deep neural networks when Caffe and Caffe plus cuDNN implementations are considered. The results are evaluated on several public datasets used in remote sensing for land-cover applications. The classification scheme (HYCNN) consists of principal component analysis, patch extraction, convolution filters and fully connected layers. The learning is performed using the standard back-propagation algorithm.

The CUDA GPU implementations considered are Caffe-HYCNN and cuDNN-HYCNN. The first one uses the Caffe functions while the second one performs calls to the cuDNN functions, showing that this second version is more efficient. The classification accuracy obtained is different for the different implementations, and is increased when two convolutional layers are considered, achieving values of up to 98.15% for the Pavia C. dataset with the cuDNN-HYCNN code. Regarding the execution times, several GPU optimization strategies are applied obtaining speedups of up to 74.22× for Pavia U. with cuDNN-HYCNN with respect to the HYCNN in CPU.

Acknowledgements This work was supported in part by the Consellería de Cultura, Educación e Ordenación Universitaria (Xunta de Galicia) [Grant Numbers GRC2014/008 and ED431G/08] and Ministry of Education, Culture and Sport, Government of Spain [Grant Numbers TIN2013-41129-P and TIN2016-76373-P]. These grants are co-funded by the European Union (European Regional Development Fund - ERDF).

References

1. Fauvel M, Tarabalka Y, Benediktsson JA, Chanussot J, Tilton JC (2013) Advances in spectral-spatial classification of hyperspectral images. Proc IEEE 101(3):652–675
2. Yue J, Zhao W, Mao S, Liu H (2015) Spectral-spatial classification of hyperspectral images using deep convolutional neural networks. Remote Sens Lett 6(6):468–477
3. Hu W, Huang Y, Wei L, Zhang F, Li H (2015) Deep convolutional neural networks for hyperspectral image classification. J Sens https://doi.org/10.1155/2015/258619
4. Makantasis K, Karantzalos K, Doulamis A, Doulamis N (2015) Deep supervised learning for hyperspectral data classification through convolutional neural networks. In Proceeding of IEEE International Geoscience and Remote Sensing Symposium (IGARSS), pp 4959–4962
5. Zhao W, Du S (2016) Spectral-spatial feature extraction for hyperspectral image classification: a dimension reduction and deep learning approach. IEEE Trans Geosci Remote Sens 54(8):4544–4554
6. Chen Y, Jiang H, Li C, Jia X, Ghamisi P (2016) Deep feature extraction and classification of hyperspectral images based on convolutional neural networks. IEEE Trans Geosci Remote Sens 54(10):6232–6251
7. Li Y, Zhang H, Shen Q (2017) Spectral-spatial classification of hyperspectral imagery with 3D convolutional neural network. Remote Sens 9(1):67
8. Christophe E, Michel J, Inglada J (2011) Remote sensing processing: from multicore to GPU. IEEE J Sel Top Appl Earth Obs Remote Sens 4(3):643–652
9. Plaza A, Du Q, Chang Y, King RL (2011) High performance computing for hyperspectral remote sensing. IEEE J Sel Top Appl Earth Obs Remote Sens 4(3):528–544
10. Jia Y, Shelhamer E, Donahue J, Karayev S, Long J, Girshick R, Guadarrama S, Darrell T (2014) Caffe: Convolutional Architecture for Fast Feature Embedding, arXiv preprint arXiv:1408.5093
11. Nvidia (2017) cuDNN. https://developer.nvidia.com/cudnn. Accessed 22 Mar 2017
12. Aptoula E, Ozdemir MC, Yanikoglu B (2016) Deep learning with attribute profiles for hyperspectral image classification. IEEE Geosci Remote Sens Lett 13(12):1970–1974
13. Kirk David B, Wen-mei W. Hwu (2016) Programming Massively Parallel Processors A Hands-on Approach, Morgan Kaufmann
14. Nvidia (2016) Whitepaper: NVIDIA Tesla P100. http://www.nvidia.com/object/tesla-p100.html. Accessed 2 Dec 2016
15. Nvidia (2015) CULA Tools. http://www.culatools.com/. Accessed 13 Jan 2015
16. MAGMA (2015) Matrix Algebra on GPU and Multicore Architectures. http://icl.cs.utk.edu/projectsfiles/magma/doxygen/. Accessed 13 Jan 2017
17. Nvidia (2015) CUDA Toolkit Documentation: CUBLAS. http://docs.nvidia.com/cuda/cublas/. Accessed 11 Jan 2017
18. Garea AS, Heras DB, Argüello F (2016) GPU classification of remote sensing images using kernel ELM and extended morphological profiles. Int J Remote Sens 37(24):5918–5935
19. Fauvel M, Tarabalka Y, Benediktsson JA, Chanussot J, Tilton JC (2013) Advances in spectral-spatial classification of hyperspectral images. Proc IEEE 101(3):652–675