Neurocomputing 398 (2020) 411–421

Contents lists available at ScienceDirect

Neurocomputing

journal homepage: www.elsevier.com/locate/neucom

Compact bilinear pooling via kernelized random projection for fine-grained image categorization on low computational power devices

Daniel López-Sánchez*, Angélica González Arrieta, Juan M. Corchado

BISITE Research Group, University of Salamanca, Spain

ARTICLE INFO

Article history:
Received 30 November 2018
Revised 9 May 2019
Accepted 20 May 2019
Available online 29 July 2019

Keywords:
Bilinear pooling
Deep learning
Random projection
Polynomial kernel

ABSTRACT

Bilinear pooling is one of the most popular and effective methods for fine-grained image recognition. However, a major drawback of Bilinear pooling is the dimensionality of the resulting descriptors, which typically consist of several hundred thousand features. Even when generating the descriptor is tractable, its dimension makes any subsequent operations impractical and often results in huge computational and storage costs. We introduce a novel method to efficiently reduce the dimension of bilinear pooling descriptors by performing a Random Projection. Conveniently, this is achieved without ever computing the high-dimensional descriptor explicitly. Our experimental results show that our method outperforms existing compact bilinear pooling algorithms in most cases, while running faster on low computational power devices, where efficient extensions of bilinear pooling are most useful.

1. Introduction

The term fine-grained recognition is generally applied to describe classification tasks with a relatively large number of very similar categories. Examples of this include animal and plant species classification [1–3], automobile and plane model identification [4,5], or scene recognition [6] among others. Such classification tasks tend to be quite challenging, partly because of the high intra-class variability they exhibit, combined with a low inter-class variability. In other words, the small variations that contain the information needed to differentiate classes can be easily overwhelmed by non-informative factors such as pose, orientation, illumination conditions, etc.

In the recent years, many different approaches have been proposed to address the challenge of fine-grained recognition, and accuracies have risen steadily [7–9]. One of the most effective and widely adopted approaches proposed in the literature is the use of bilinear Convolutional Neural Networks (CNN) [10–14], originally introduced by Lin et al. [15]. In essence, bilinear CNNs build an image feature descriptor by first applying two CNNs as feature extractors. Then, the two descriptors generated are combined at each location by using the outer product. Finally, the resulting descriptor is pooled across locations to obtain a global descriptor of the input image. A classical linear classifier (e.g., softmax, logistic regression, etc.) is then applied on the global descriptor. This approach enables bilinear CNNs to capture pairwise feature interactions in a location-invariant manner, which enables a boost in fine-grained classification accuracies.

In spite of its success, the bilinear CNN approach has a major drawback. As a consequence of using the outer product, the generated descriptor is extremely high dimensional. For instance, the bilinear descriptor used in [15] had more than 250,000 features. As a consequence, even a simple linear classifier trained on these descriptors will have millions of parameters, or even hundreds of millions if the number of classes is large. This results in high computation and storage costs, and makes models more prone to over-fitting. While these heavy models might work well when deployed on powerful and specialized hardware, their applicability is severely limited on devices with little computational resources. For example, devices with limited memory might have problems to allocate space for the large number of parameters of bilinear CNNs, and the high number of operations required to process an image might result in important inference-time delays when running on devices without the massive parallelism of modern GPUs. At the same time, the emergence of new computation paradigms such as the Edge computing [16] has originated a growing need for effective machine learning models capable of running on low computational power devices such as Internet of Things (IoT) devices or embedded systems. For instance, deep learning methods have been profusely applied in tasks related to animal species recognition [17] or face recognition [10]. Due to the nature of these tasks,

* Corresponding author.
E-mail address: lope@usal.es (D. López-Sánchez).

https://doi.org/10.1016/j.neucom.2019.05.104
0925-2312/

high performance hardware might not be available at the location where models need to be deployed, which explains the need for efficient variants of the existing fine-grained image understanding methods.

Recently, methods that try to compress the discriminative information of the bilinear descriptor into low dimension representations have been developed, seeking to mitigate the efficiency problems of bilinear pooling. Most notably, Gao et al. proposed compact bilinear pooling [18], which uses polynomial kernel feature approximation techniques to achieve this. In addition, the authors of [18] also discussed the possibility of using Random Projection [19] to reduce the dimension of the bilinear feature descriptor, but discarded this idea after noting that such approach would require storing a huge projection matrix and explicitly computing the bilinear descriptor prior to the projection.

In this paper, we further develop the idea of using Random Projection to reduce the dimension of the bilinear CNN descriptor. In particular, we propose adapting an existing kernelized variant of Random Projection [20] to efficiently project bilinear descriptors to a lower dimension without ever having to explicitly compute the high-dimensional bilinear descriptor itself. By implicitly computing a Random Projection of the bilinear descriptor, our method generates a low-dimensional feature vector that captures much of the discriminative information of the full bilinear descriptor, while resulting in models with a much lower number of parameters. We also derive back-propagation for our algorithm, so that it can be included as a building block in end-to-end trainable models. As a practical application of the proposed approach, we study the task of fine-grained image classification on low computational power devices of the Raspberry Pi ecosystem. We focus on this application scenario because, as pointed out by Gao et al. [18], methods for making bilinear pooling more efficient are most useful in low power devices such as embedded systems, where computational resources are scarce. Our experimental results show that the proposed algorithm generates a better compacted representation of the bilinear descriptor in most cases, while being notably faster than alternative compact bilinear pooling approaches.

2. Related work

Bilinear models were originally proposed in [21], where the authors used them to separately model the style and content of images. More recently, Lin et al. [15] explored their applicability in the context of deep learning [17,22] for fine-grained image categorization, showing that bilinear Convolutional Neural Networks could be used to achieve state of the art results in various fine-grained image categorization datasets.

In [18], the authors applied two polynomial kernel approximation techniques to make bilinear CNNs less computationally demanding, especially in terms of the memory required for parameter storage. This approach emerged from the notion that bilinear features are fundamentally related to the feature space of the homogeneous polynomial kernel of degree two, so kernel approximation feature maps can also be used to approximate bilinear pooling descriptors. This approach is known as compact bilinear pooling, since it reduces the dimension of the bilinear descriptor proposed in [15]. In addition, back-propagation was derived for both methods in [18], making the proposed models end-to-end trainable.

The first kernel approximation technique applied in [18] was Random Maclaurin [23] (RM). In essence, Random Maclaurin builds a randomized feature map which, when approximating the degree-two homogeneous polynomial kernel, takes the form $Z : \mathbb{R}^d \to \mathbb{R}$, Z: $\mathbf{x} \to \langle \mathbf{x}, \mathbf{w}_1 \rangle \langle \mathbf{x}, \mathbf{w}_2 \rangle$ where $\mathbf{x} \in \mathbb{R}^d$ is the input data sample and $\mathbf{w_1}, \mathbf{w_2} \in \mathbb{R}^d$ are i.i.d. random Rademacher vectors. Conveniently, for two arbitrary data samples $\mathbf{x}, \mathbf{y}$, it can be proven that $\mathbb{E}[Z(\mathbf{x})Z(\mathbf{y})] = \langle \mathbf{x}, \mathbf{y} \rangle^2$. Of course, the quality of this feature map can be improved by using more than one entries in the output representation, thus reducing the variance of the estimator. While this approach performed well in the experiments of [18], it has the inherent limitation of requiring a significant amount of memory to store the Rademacher vectors used for the map.

The second kernel approximation technique used in [18] was Tensor Sketch [24] (TS). Introduced a few years later than Random Maclaurin, Tensor Sketch obtains a Count Sketch [25] of the outer product of two vectors in an efficient manner, which can be used to approximate polynomial kernels and in turn the bilinear descriptor. In particular, instead of explicitly computing the outer product, TS computes the Count Sketch of the vectors and then uses polynomial multiplication via the Fast Fourier Transform to compute the Count Sketch of their outer product. Using this method to achieve a compact bilinear pooling typically results in higher accuracies, while requiring much less memory for parameter storage than RM.

In addition to compact bilinear pooling methods, low-rank matrix factorization methods have also been proposed to make bilinear pooling more efficient, mainly by avoiding the high-dimensionality of the full bilinear descriptor, which leads to models with too many trainable parameters. In particular, Kim et al. [26] proposed a low-rank bilinear pooling method based on the Hadamard product. In essence, this approach is based on the factorization a three-dimensional weight tensor applied to the bilinear descriptor into three two-dimensional matrices, greatly reducing the number of parameters to be learned. Conveniently, low-rank bilinear pooling can be easily adapted to multimodal learning models, such as those used for Visual Question Answering (VQA). In fact, a recent study by Yu et al. [27] generalized multimodal factorized bilinear pooling to capture high-order interactions between multi-modal features, obtaining a state-of-the-art performance on two large-scale real-world VQA datasets.

As mentioned above, Gao et al. [18] also suggested the possibility of using Random Projections (RP) [19] to reduce the dimensionality of bilinear descriptors. Thanks to the Johnson–Lindenstrauss lemma [28] that underpins Random Projection, pairwise distances between bilinear descriptors would be approximately preserved in the projected representation. However, they discarded this idea because directly applying RP to the bilinear descriptors would involve storing a large projection matrix and explicitly computing the bilinear descriptors in the first place. However, recent advances in the intersection of kernel methods and Random Projection [20,29–31] have made it possible to efficiently perform Random Projections from the feature spaces of different kernel functions in an efficient manner. In particular, an efficient method to approximate a Random Projection for polynomial kernels was introduced in [20]. This paper adapts the ideas presented in [20] to make bilinear CNNs less computationally demanding by approximating a Random Projection of the bilinear descriptor.

3. Proposed approach

Bilinear pooling [15] computes a global descriptor for an image $\mathcal{I}$ by computing the outer product of local descriptors and then applying average pooling over locations. In the context of this paper, the local descriptors are generated by means of an arbitrary CNN (see Fig. 1). Formally, the global bilinear descriptor is defined as:

$$\Phi(\mathcal{I}) = \sum_{l \in \mathcal{L}} \text{CNN}(\mathcal{I}, l) \otimes \text{CNN}(\mathcal{I}, l), \tag{1}$$

where $\text{CNN}(\mathcal{I}, l)$ denotes the descriptor extracted from image $\mathcal{I}$ at location l by the chosen CNN[1], $\mathcal{L}$ is the set of existing locations

[1] Note that, like in [18], we focus on the case were the same feature-extraction CNN is used in both sides of the Kronecker product.

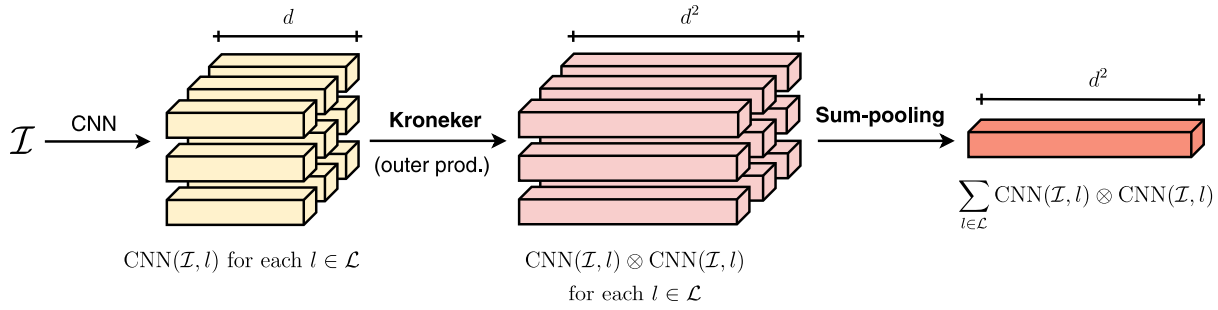

Fig. 1. Schematic view of Bilinear Pooling [15] for an input image $\mathcal{I}$ and a Convolutional Neural Network (CNN) which produces an output feature map with d channels. First, the Kronecker product is applied at each location of the feature maps generated by the CNN. Then, the resulting bilinear descriptors are averaged to form the final global bilinear descriptor.

and $\otimes$ denotes the Kronecker product.[2] For instance, if the CNN generates feature maps of dimension $H\times W$ with d channels, there will be HW locations in $\mathcal{L}$, and each local descriptor $\mathrm{CNN}(\mathcal{I},l)$ will be of size d. As a consequence, the final bilinear descriptor $\Phi(\mathcal{I})$ will be of dimension d^2, which is the main cause of the inefficiency of this approach. The descriptor is typically normalized by first applying an element-wise signed square root operation (i.e., $x\leftarrow \mathrm{sgn}(x)\sqrt{|x|}$), followed by L2 normalization.

To mitigate the issue of the high dimensionality of the bilinear descriptor, one possible approach is to perform a Random Projection to reduce its dimension. In practice, performing the Random Projection consists in multiplying the bilinear descriptor $\Phi(\mathcal{I})\in\mathbb{R}^{d^2}$ by a projection matrix $R\in\mathbb{R}^{d^2\times k}$ whose entries are independently drawn from a suitable distribution, and then applying a scaling factor to compensate for the reduction of dimensionality [19]. Formally, the Random Projection of the bilinear descriptor is:

$$\frac{1}{\sqrt{k}}\Phi(\mathcal{I})R=\frac{1}{\sqrt{k}}\left(\sum_{l\in\mathcal{L}}\mathrm{CNN}(\mathcal{I},l)\otimes\mathrm{CNN}(\mathcal{I},l)\right)R, \tag{2}$$

which results in a k-dimensional descriptor. Intuitively, we can think of each output feature from this operation as the projection of the bilinear descriptor onto one of the columns of the projection matrix. Regarding the distribution for the entries of R, several options have been proposed throughout the years. Originally, uniform and standard normal distributions were used [32,33]. Later on, studies demonstrated that projection matrices can be drawn from much simpler distributions. For instance, Achlioptas showed that the entries of the projection matrix can be instead drawn from a discrete and sparse distribution [19]. In particular, Achlioptas' work proved that if the entries of R are drawn from the distribution defined by (3) with sparsity term $s=1$ or $s=3$, then the result will be a valid Random Projection.

$$r_{ij}=\sqrt{s}\begin{cases}1 & \text{with prob. } 1/2s,\\ 0 & \text{with prob. } 1-1/s,\\ -1 & \text{with prob. } 1/2s.\end{cases} \tag{3}$$

Therefore, we can see than Random Projection is a rather robust method in terms of the required distribution for the projection matrix, since many distributions can be used. A crucial point however is that, regardless of the selected distribution, the entries of the projection matrix must be chosen independently.

It is worth noting that using the distribution proposed by Achlioptas reduces the computational cost of the projection. If the multiplication by $\sqrt{s}$ present in (3) is delayed, the computation of the projection itself reduces to aggregate evaluation (i.e. summation and subtraction but no multiplication), which can be efficiently performed in database environments using standard SQL primitives. In addition, the sparsity term s enables further storage and computational savings. For instance, when using $s=3$, only $\frac{1}{3}$ of the entries of the projection matrix are nonzero. Moreover, it has been suggested that using greater sparsity levels in (3) is possible with little loss in accuracy. In particular, some studies recommend using $s=\mathcal{O}(\sqrt{d})$ [34].

However, as pointed out by Gao et al. [18], even if a sparse random projection matrix is used, the d^2-dimensional bilinear descriptor needs to be computed before performing the projection in (2), incurring in much of the inefficiencies of standard bilinear pooling. Luckily, various methods have been recently introduced to efficiently perform Random Projections from kernel feature spaces. In particular, the kernelized algorithm proposed in [20] can be used to perform Random Projections from the feature space of homogeneous polynomial kernels of degree two. Moreover, the same ideas can be used to apply a Random Projection to bilinear descriptors in an efficient manner. To show this, we begin by examining the following property of the Kronecker product. Let $\mathbf{x},\mathbf{r}_1,\mathbf{r}_2\in\mathbb{R}^d$ be three arbitrary vectors. Then the following holds:

$$\langle\mathbf{x},\mathbf{r}_1\rangle\langle\mathbf{x},\mathbf{r}_2\rangle=\langle\mathbf{x}\otimes\mathbf{x},\ \mathbf{r}_1\otimes\mathbf{r}_2\rangle. \tag{4}$$

This equality is used in [20] to perform operations in the feature space of homogeneous polynomial kernel without ever computing it explicitly. Note that $\phi(\cdot)$: $\mathbf{x}\rightarrow\mathbf{x}\otimes\mathbf{x}$ is a valid feature map for the homogeneous polynomial kernel of degree two, so the inner product in the right hand side of the above equation can be thought as taking place in the feature space of that kernel. Conveniently, the inner products in the left hand side of the equation are in $\mathbb{R}^d$, which enables us to evaluate the expression in an efficient manner.

At this point, we might attempt to exploit (4) to perform a Random Projection of $\mathbf{x}\otimes\mathbf{x}$, as a first step towards our goal of projecting the bilinear descriptor. To achieve this, $\mathbf{r}_1$ and $\mathbf{r}_2$ should be chosen in such a way that the entries of $\mathbf{r}_1\otimes\mathbf{r}_2$ follow one of the valid Random Projection distributions discussed before, so $\mathbf{r}_1\otimes\mathbf{r}_2$ can play the role of one of the columns of the projection matrix. For instance, if we draw $\mathbf{r}_1$ and $\mathbf{r}_2$ according to (3) with $s=1$, then the entries of $\mathbf{r}_1\otimes\mathbf{r}_2$ will appear to also follow this distribution when analyzed individually. However, the entries of $\mathbf{r}_1\otimes\mathbf{r}_2$ are not mutually independent, which as mentioned before is a crucial requirement for achieving a Random Projection.

As shown in [20], one possible solution to overcome this problem is to apply the multidimensional Central Limit Theorem (CLT) [35]. This classical result states that the sum of t i.i.d. random vectors with zero means and $\boldsymbol{\Sigma}$ covariance, scaled by $1/\sqrt{t}$, converges in distribution to a multivariate normal with zero means and $\boldsymbol{\Sigma}$ covariance as t goes to infinity. As a consequence, given $2t$ i.i.d.

[2] We use the Kronecker product rather than the outer product to characterize bilinear pooling for the sake of consistency with the notation in [20], but these operations are essentially equivalent in this case.

zero-mean random vectors $\mathbf{r}_1, \ldots, \mathbf{r}_{2t}$, we can ensure that

$$\sum_{j=0}^{t-1} \left(\frac{\mathbf{r}_{2j+1} \otimes \mathbf{r}_{2j+2}}{\sqrt{t}} \right) \tag{5}$$

converges in distribution to a multidimensional normal distribution with zero means. Moreover, if vectors we are summing have identity covariance matrix, then (5) converges in distribution to a multidimensional normal with zero means and identity covariance, which is one of the valid distributions for the Random Projection matrix [33]. Conveniently, the desired identity covariance for vectors in the summation of (5) can be achieved by independently drawing the entries of $\mathbf{r}_1, \ldots, \mathbf{r}_{2t}$ from Achlioptas' distribution, displayed in (3). Note that, by definition, the individual variables in a multidimensional normal with identity covariance are independent, so the dependence among the entries of vectors formed following (5) vanishes as t grows.

Therefore, if we use projection vectors generated following (5), with $\mathbf{r}_1, \ldots, \mathbf{r}_{2t}$ populated according to (3), then for a sufficiently large t we will be performing a valid Random Projection. Formally, each component $\mathbf{y}_i$ of the output representation will be:

$$\mathbf{y}_i = \frac{1}{\sqrt{k}} \left\langle \Phi(\mathcal{I}), \sum_{j=0}^{t-1} \left(\frac{\mathbf{r}_{2j+1} \otimes \mathbf{r}_{2j+2}}{\sqrt{t}} \right) \right\rangle_{\mathbb{R}^{d^2}}. \tag{6}$$

As shown in [20], even if the selected value of t is not big enough to make the resulting projection vectors follow a perfect normal distribution, the summation in (5) has the effect of reducing the statistical dependence among the entries of the projection vectors, resulting in a better approximation of a proper Random Projection.

However, directly using (6) to compute the RP of the bilinear descriptor involves explicitly generating the descriptor and the projection vectors, resulting in the same inefficiencies as directly applying standard RP. Luckily, the inner product of the bilinear descriptor and our projection vectors can be conveniently rewritten to avoid working in the d^2-dimensional space. This is achieved by using (4) along with some elemental properties of inner products:

$$\begin{aligned} \mathbf{y}_i &= \frac{1}{\sqrt{k}} \left\langle \Phi(\mathcal{I}), \sum_{j=0}^{t-1} \left(\frac{\mathbf{r}_{2j+1} \otimes \mathbf{r}_{2j+2}}{\sqrt{t}} \right) \right\rangle \\ &= \frac{1}{\sqrt{tk}} \sum_{j=0}^{t-1} \left\langle \Phi(\mathcal{I}), \quad \mathbf{r}_{2j+1} \otimes \mathbf{r}_{2j+2} \right\rangle \\ &= \frac{1}{\sqrt{tk}} \sum_{l \in \mathcal{L}} \sum_{j=0}^{t-1} \left\langle \mathrm{CNN}(\mathcal{I}, l) \otimes \mathrm{CNN}(\mathcal{I}, l), \ \mathbf{r}_{2j+1} \otimes \mathbf{r}_{2j+2} \right\rangle \\ &= \frac{1}{\sqrt{tk}} \sum_{l \in \mathcal{L}} \sum_{j=0}^{t-1} \left\langle \mathrm{CNN}(\mathcal{I}, l), \mathbf{r}_{2j+1} \right\rangle \left\langle \mathrm{CNN}(\mathcal{I}, l), \mathbf{r}_{2j+2} \right\rangle. \end{aligned} \tag{7}$$

Conveniently, the inner products appearing in the last expression are in $\mathbb{R}^d$, avoiding the need of explicitly computing the bilinear descriptor and the d^2-dimensional projection vectors. The complete output representation generated by our algorithm is obtained by repeating this projection k times, each with a different set of vectors $\mathbf{r}_1, \ldots, \mathbf{r}_{2t}$:

$$\mathbf{y} = [\mathbf{y}_1, \ldots, \mathbf{y}_k]. \tag{8}$$

Regarding the selection of the hyperparameter t, the results in [20] suggest that while relatively high values of t are required for good pairwise-distance preservation after the projection, classification accuracies do not benefit much from using values of t greater than two. In fact, the authors recommended using small values of t in classification scenarios to reduce the computational cost.

3.1. Reusing vectors for improved efficiency

Up to this point, we have assumed that each of the output components $\mathbf{y}_i$ of the representation generated by our algorithm uses a completely different set of vectors $\mathbf{r}_1, \ldots, \mathbf{r}_{2t}$. This ensures that the projection vectors generated using (5) for different output components are independent of each other, which is required to achieve a valid Random Projection. Unfortunately, this also forces us to maintain a total of $2tk$ d-dimensional vectors in memory, which in some cases can be challenging. However, as shown in [20], this requirement can be relaxed in practice. In particular, instead of using $2tk$ different vectors, the authors of [20] proposed generating a set $S = \{\mathbf{r}_1, \ldots, \mathbf{r}_p\}$ containing p i.i.d. vectors, and then using a random subset $S_i \subset S$ for each output component. This approach produced good results in practice, while enabling substantial computational savings [20].

A similar approach is taken in this paper. First, a set containing p i.i.d. random vectors $S = \{\mathbf{r}_1, \ldots, \mathbf{r}_p\}$ is generated with the entries of each vector following the distribution defined in (3) and $2t < p \leq 2tk$. Then, $2t$ of those vectors are selected for each output component $\mathbf{y}_1, \ldots, \mathbf{y}_k$. However, rather than simply selecting k random subsets of S as done in [20], we make sure that each individual vector $\mathbf{r}_i$ is used the lowest number of times possible. In contrast, randomly selecting k subsets of S results in some vectors being used more often than others. Also, a particular vector might not be used at all. To achieve a more even usage of the vectors in S, we first generate a collection P containing the elements of S repeated the necessary number of times to ensure $|P| = 2tk$:

$$P = \underbrace{S \cup \ldots \cup S}_{\lfloor 2tk/p \rfloor} \cup S[1 : 2\mathrm{tk} \bmod \mathrm{p}]. \tag{9}$$

Then, we sample P without replacement to form k collections $S_1, \ldots, S_k$ each with $2t$ vectors. The $2t$ vectors in collection S_i are then used in the computation of the output component $\mathbf{y}_i$, using (7). In practice, $S_1, \ldots, S_k$ store references to the original vectors is S rather than copies of them, so no extra memory needs to be allocated. Algorithm 1 provides a self-contained high-level description of the proposed method. Throughout the following sections, we will refer to this algorithm as Compact Bilinear Pooling via Kernelized Random Projection (CBP-KRP).

3.2. Computational complexity and implementation tricks

Analyzing the different steps in Algorithm 1, it is possible to determine both the time complexity and storage requirements of the proposed method. Steps 1–3 correspond to the instantiation of the algorithm, and contain the initialization of the parameters of the model. Most of the memory cost comes from storing S, which contains p vectors of dimension d. Luckily, these vectors are drawn from Achlioptas' sparse distribution, so using an appropriate sparse matrix implementation the zero-valued entries need not be stored. Therefore, only $\mathcal{O}(dp/s)$ parameters need to be stored to represent S.

Regarding the collections P and $S_1, \ldots, S_k$, as mentioned before, they can be implemented in such a way that they only store references to the original vectors in S, so the memory requirements are reduced significantly. In addition, note that P is only temporarily used to form $S_1, \ldots, S_k$. In total, $S_1, \ldots, S_k$ contain $2tk$ references[3] that need to be stored after the initialization of the algorithm, together with the set of vectors S. Therefore, the complete model requires storing $\mathcal{O}(dp/s) + 2tk$ parameters.

[3] Depending on the implementation, these references can take the form of integer indexes, memory pointers, etc. In any case, storing one of these references has a similar memory cost as storing a floating point parameter.

Algorithm 1 Compact Bilinear Pooling via Kernelized Random Projection (CBP-KRP).

Require: Descriptors $\text{CNN}(\mathcal{I}, l)$ for some image $\mathcal{I}$ at each location $l \in \mathcal{L}$. The total number of vectors p and their sparsity level s, the number t of vectors used for the Central Limit Theorem and the desired output dimension k.

Ensure: Returns a k-dimensional vector which approximates a Random Projection of the full bilinear pooling descriptor.

1: $S \leftarrow \{\mathbf{r}_1, \ldots, \mathbf{r}_p\}$ where each entry of $\mathbf{r}_i \in \mathbb{R}^d$ is $\{-\sqrt{s}, 0, \sqrt{s}\}$ w.p. $\{\frac{1}{2s}, 1-\frac{1}{s}, \frac{1}{2s}\}$ ▷ Generate vectors

2: $P = \underbrace{S \cup \ldots \cup S}_{\lfloor 2tk/p \rfloor} \cup S[1 : 2\text{tk mod p}]$, so that $|P| = 2tk$ ▷ Generate a redundant collection to sample from

3: Sample P w/o replacement to form $S_1, \ldots, S_k$, where $|S_i| = 2t$ ▷ Select $2t$ vectors for each output dimension

4: $\mathbf{y} \leftarrow [0, \ldots, 0] \in \mathbb{R}^k$ ▷ Initialize output vector

5: **for** $l \in \mathcal{L}$ **do** ▷ Iterate over each location

6: **for** $i = 1, \ldots, k$ **do** ▷ Iterate over each output dimension

7: **for** $j = 0, \ldots, t-1$ **do**

8: $\mathbf{y}_i \leftarrow \mathbf{y}_i + \frac{1}{\sqrt{t}} \langle \text{CNN}(\mathcal{I}, l), S_i[2j+1] \rangle \cdot \langle \text{CNN}(\mathcal{I}, l), S_i[2j+2] \rangle$ ▷ Apply Eq. (**??**) to compute the projection

9: $\mathbf{y} \leftarrow \frac{1}{\sqrt{k}} \cdot \mathbf{y}$ ▷ Scale to compensate for the dimensionality reduction

10: **return** $\mathbf{y}$

To assess the computational complexity, we separately consider the initialization phase (steps 1–3) and the projection of the bilinear descriptor (steps 4–10). Regarding the initialization, the computational cost is $\mathcal{O}(dp + tk)$, where the $\mathcal{O}(dp)$ comes from forming S and the $\mathcal{O}(tk)$ from the sampling of P to form $S_1, \ldots, S_k$. In practice, these initialization steps only have to be executed once and require a time in the order of seconds at most.

For the projection of the bilinear descriptor (steps 4–10), a more detailed analysis is required. As we can see, these steps consist of a series of nested loops that, as the innermost operation, perform two inner products between vectors of dimension d. Therefore, considering the number of iterations of each loop and the cost of these inner products, we can conclude that the complexity of these steps is $\mathcal{O}(Lktd)$, where L is the number of local descriptors $L = |\mathcal{L}|$. However, one may notice that most of the inner products computed are redundant, since $S_1, \ldots, S_k$ only contain references to p unique vectors and we are computing $2tk$ inner products for each local descriptor $\text{CNN}(\mathcal{I}, l)$. As shown in [20], a much more efficient strategy would be precomputing the inner products of the L local descriptors with the p vectors in S before steps 4–5. With these inner products precomputed, the expression in step 8 can be evaluated in $\mathcal{O}(1)$ time. Therefore, applying this implementation trick the total time complexity of the proposed algorithm simplifies to $\mathcal{O}(L(pd + tk))$, where the $\mathcal{O}(Lpd)$ comes from precomputing the inner products and the $\mathcal{O}(Ltk)$ from executing steps 4–10. It is also important to note that, thanks to the sparse nature of the vectors in S, the computation of the inner products can be accelerated by using sparse matrix multiplication routines, available in most linear algebra packages.

Table 1 compares the number of parameters and time complexity of the proposed method with the full bilinear descriptor [15] and with existing compact bilinear pooling methods [18]. In addition to the number of parameters needed to compute the final descriptor in each case, the table also shows the number of parameters of a one-vs-all linear classifier trained on the resulting descriptor, which in the case of the full bilinear descriptor is the main source of inefficiency. Some empirical values obtained for the particular hyperparameters and CNNs used in Section 4 are also provided.

3.3. Back-propagation for CBP-KRP

One of the main features of existing compact bilinear pooling methods [18] is their compatibility with the back-propagation algorithm, which makes them end-to-end trainable. The fact that the partial derivative of the output of these algorithms with respect to their input can be easily computed makes it possible to include them as intermediate layers in deeper models, as the gradient of the loss function can be back-propagated towards the first layers using the chain rule.

In this section, we derive back-propagation for the proposed method, thus showing that it is also compatible with end-to-end training. First, let $\mathfrak{L}$ denote the selected loss function. To keep the notation simple, we will denote the local descriptor $\text{CNN}(\mathcal{I}, l)$ as

Table 1
Comparison of descriptor dimension, memory usage and time complexity for the different approaches and networks considered in this paper. Variables d, L and c represent the number of channels before the pooling operation, the number of locations at which the CNN is applied (i.e., height times width of the feature maps), and the number of classes respectively. Hyperparameter k corresponds to the desired output dimension for CBP-KRP, TS and RM. Hyperparameters t, p and s control the behavior of CBP-KRP (see Algorithm 1). Numeric results are for $c = 200$, $k = 5000$, $p = 5000$, $t = 2$ and $s = 100$, using *float32* precision.

		Full Bilinear	CBP-KRP	TS [18]	RM [18]
Theoretic	Descriptor Size	d^2	k	k	k
	Parameters	0	$\mathcal{O}(dp/s) + 2tk$	$4d$	$2dk$
	Classifier Param.	cd^2	ck	ck	ck
	Computation	$\mathcal{O}(Ld^2)$	$\mathcal{O}(L(pd + tk))$	$\mathcal{O}(L(d + k\log k))$	$\mathcal{O}(Ldk)$
SqueezeNet [36]	Descriptor Size	**262,144**	5,000	5,000	5000
Network size: 4.8 MB	Parameters	0 B	280 KB	8 KB	**19.5 MB**
@ *fire9* ($13 \times 13 \times 512$)	Classifier Param.	**200 MB**	3.8 MB	3.8 MB	3.8 MB
GoogLeNet [37]	Descriptor Size	**692,224**	5,000	5,000	5000
Network size: 25.7 MB	Parameters	0 B	406 KB	13 KB	**31.7 MB**
@ *incept-4e* ($14 \times 14 \times 832$)	Classifier Param.	**528 MB**	3.8 MB	3.8 MB	3.8 MB

Table 2
Main features of the two Raspberry devices used for the inference-time experiments.

Raspberry Model	CPU model	CPU Cores & Freq.	RAM	Release	Price
Pi 3 Model B+	BCM2837B0 (Cortex-A53)	4 @ 1.4 GHz	1 GB	14/03/18	$35
Pi Zero W	BCM2835 (ARM1176JZF-S)	1 @ 1 GHz	512 MB	28/02/17	$10

$\mathbf{x}_l$. Therefore, the input to the proposed algorithm is the set of d-dimensional local descriptors $\{\mathbf{x}_l\}_{l\in\mathcal{L}}$. The output of the algorithm is the k-dimensional projection $\mathbf{y}\in\mathbb{R}^k$ of the bilinear descriptor. Back-propagation for our algorithm can then be written as follows:

$$\frac{\partial \mathfrak{L}}{\partial \mathbf{x}_l} = \sum_{i=1}^{k} \frac{\partial \mathfrak{L}}{\partial \mathbf{y}_i} \frac{\partial \mathbf{y}_i}{\partial \mathbf{x}_l}, \tag{10}$$

$$\frac{\partial \mathbf{y}_i}{\partial \mathbf{x}_l} = \frac{1}{\sqrt{tk}} \sum_{j=0}^{t-1} \Big(\langle \mathbf{x}_l, S_i[2j+1] \rangle S_i[2j+2] + \langle \mathbf{x}_l, S_i[2j+2] \rangle S_i[2j+1] \Big).$$

The first equability is derived by simply applying the chain rule, and the second one is the partial derivative of the ith feature in $\mathbf{y}$ with respect to one of the local descriptors $\mathbf{x}_l$. With this two equations, one can propagate the gradient of the loss function across our algorithm to layers closer to the input. While it might be possible to derive the gradient with respect to the vectors in $S_1, \ldots, S_k$ to also update them during training, this is not recommended because (1) the sparsity of the vectors would be lost, and (2) we would no longer be approximating a Random Projection of the bilinear descriptor, as the distribution of the projection vectors would be altered. Section 4.4 presents experimental results on the fine-tuning of CNNs with the proposed algorithm as an intermediate layer.

4. Experimental results and discussion

In this section, we present experimental results regarding both the efficiency and accuracy achieved by the proposed algorithm as compared with existing approaches. As mentioned before, our inference-time results focus on low computational power devices. As shown in [18], when compact bilinear pooling is executed in specialized hardware such as GPUs, the high level of parallelism in such devices makes bilinear pooling reasonably fast, to the point that compact bilinear pooling can be even slower.[4] In addition, the dominant factor in most cases is the forward pass of the convolution layers, so improvements in the efficiency of bilinear pooling might not have a significant impact in the total inference time of the entire model. In such scenarios, the main advantage of compact bilinear pooling methods is the reduction in the number of parameters of the model, as a consequence of the reduced dimensionality of the descriptor. Conversely, when running on low computational power devices, compact bilinear pooling methods can make a huge difference both in terms of memory requirements and total inference times.

We perform inference-time experiments on two devices from one of most widespread low-cost hardware platforms. In particular, we used the Raspberry Pi 3 Model B+, the latest version of the classic Raspberry series, and the Raspberry Pi Zero W, the smallest Raspberry computer.[5] Table 2 highlights some of the most important features of these devices, and Fig. 2 shows their relative sizes. Given their widespread use, some of the most popular deep learning platforms such as Tensorflow [38] now include support for

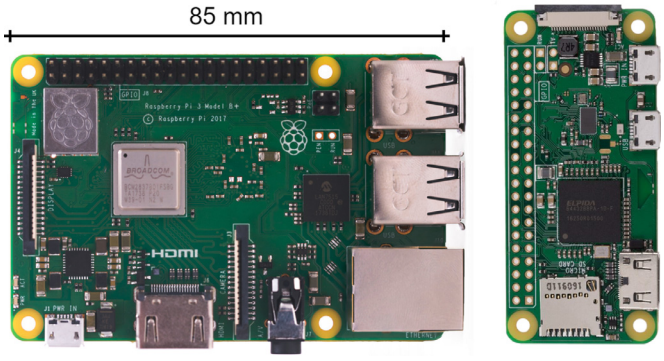

Fig. 2. Raspberry Pi Model 3 B+ (left) and Zero W (right).

installation on devices of the Raspberry ecosystem. This reflects the growing interest of the community in running deep learning models on low cost and low power devices.

4.1. Evaluated methods

Since our experiments focus on inference-time in low power devices, we selected two relatively lightweight pretrained CNNs to make sure that the models would fit in memory. In particular, we used SqueezeNet v1.1 [36] and GoogLeNet [37] CNNs. On the one hand, SqueezeNet is a recently proposed architecture specifically designed for efficiency. Notably, the weights of this CNN only require 4.8 MB of storage, and even less if weight compression techniques are applied. Version v1.1 of this model achieves a similar accuracy as the original one while being twice as fast.[6] On the other hand, GoogLeNet is a slightly heavier model with a size of 25.7 MB, which was the winning architecture on the ImageNet 2014 challenge. Conveniently, public implementations exist for both models[7], based on the Keras [39] and Tensorflow [38] Python libraries.[8] Tables 3 and 4 provide a detailed description of the architectures of these CNNs. The cut-off layers at which the bilinear pooling operation was performed were *fire9* for SqueezeNet and *inception (4e)* for GoogLeNet. The different evaluated approaches were as follows:

- Baseline: The CNNs model is chopped at the specified cut-off layer. Then, a signed square root operation is applied followed by L2 normalization of the features. A one-vs-rest linear SVM classifier [40] is then trained directly on these features.
- Full bilinear pooling (FB): The CNN model is chopped at the specified cut-off layer. Then, the bilinear pooling descriptor is generated [15] for the feature maps at the cut-off layer, followed by a signed square root operation and L2 normalization. A one-vs-rest linear SVM classifier [40] is then trained on the full bilinear descriptors.
- Compact bilinear pooling via Kernelized Random Projection (CBP-KRP): The CNN model is chopped at the specified cut-off layer. Then, Algorithm 1 is applied on the feature maps at the

[4] For instance, [18] reported that full bilinear pooling and TS compact bilinear pooling required 0.77 ms and 5.03 ms respectively, while the time for a pass of the CNN they used was 312 ms.

[5] https://www.raspberrypi.org/products/.

[6] https://github.com/DeepScale/SqueezeNet.

[7] https://github.com/rcmalli/keras-squeezenet https://github.com/fchollet/deep-learning-models/pull/59.

[8] We used Keras version 2.1.1 and Tensorflow version 1.9.0 in all our experiments.

Table 3
Overview of the architecture of the SqueezeNet v1.1 CNN. For more details about the architecture and custom layers used by SqueezeNet see the original publication [36] and the official repository at https://github.com/DeepScale/SqueezeNet.

Layer	Filter/stride	Output Size	Depth
input image	–	227 × 227 × 3	–
conv1	3 × 3/2 (× 64)	113 × 113 × 64	1
maxpool1	3 × 3/2	56 × 56 × 64	0
fire2	–	56 × 56 × 128	2
fire3	–	56 × 56 × 128	2
maxpool3	3 × 3/2	27 × 27 × 128	0
fire4	–	27 × 27 × 256	2
fire5	-	27 × 27 × 256	2
maxpool5	3 × 3/2	13 × 13 × 256	0
fire6	–	13 × 13 × 384	2
fire7	–	13 × 13 × 384	2
fire8	–	13 × 13 × 512	2
fire9	–	13 × 13 × 512	2
conv10	1 × 1/1 (× 1000)	13 × 13 × 1000	1
avgpool10	13 × 13/1	1 × 1 × 1000	0

Table 4
Overview of the architecture of the GoogLeNet CNN. For more details about the architecture and custom layers used by GoogLeNet see the original publication [37].

Layer	Filter/stride	Output Size	Depth
input image	–	224 × 224 × 3	–
conv1	7 × 7/2 (× 64)	112 × 112 × 64	1
maxpool1	3 × 3/2	56 × 56 × 64	0
conv2	3 × 3/1 (× 192)	56 × 56 × 192	2
maxpool2	3 × 3/2	28 × 28 × 192	0
inception (3a)	–	28 × 28 × 256	2
inception (3b)	–	28 × 28 × 480	2
maxpool3	3 × 3/2	14 × 14 × 480	0
inception (4a)	–	14 × 14 × 512	2
inception (4b)	–	14 × 14 × 512	2
inception (4c)	–	14 × 14 × 512	2
inception (4d)	–	14 × 14 × 528	2
inception (4e)	–	14 × 14 × 832	2
maxpool4	3 × 3/2	7 × 7 × 832	0
inception (5a)	–	7 × 7 × 832	2
inception (5b)	–	7 × 7 × 1024	2
avgpool5	7 × 7/1	1 × 1 × 1024	0
softmax	–	1 × 1 × 1000	1

cut-off layer to compute a compact version of the bilinear descriptor, followed by a signed square root operation and L2 normalization. A one-vs-rest linear SVM classifier [40] is trained on the resulting descriptors. For CBP-KRP, unless stated otherwise we used $p = 5000$, $t = 2$ and $s = 100$. The algorithm itself was implemented in Python, using the standard linear algebra libraries [41] and numba [42] to accelerate loops where possible.

- Compact bilinear pooling via Random Maclaurin (RM): The CNN model is chopped at the specified cut-off layer. Random Maclaurin [18,23] is used to generate a compact representation of the outer product of each local descriptor, and the resulting descriptors are average-pooled. Then, a signed square root operation is applied, followed by L2 normalization. A one-vs-rest linear SVM classifier [40] is then trained on the resulting descriptors. The original Matlab implementation of RM was rewritten in Python, using the standard linear algebra libraries [41].
- Compact bilinear pooling via Tensor Sketch (TS): The CNN model is chopped at the specified cut-off layer. Tensor Sketch [18,24] is used to generate a compact representation of the outer product of each local descriptor, and the resulting descriptors are average-pooled. Then, a signed square root operation is applied, followed by L2 normalization. A one-vs-rest linear SVM classifier [40] is then trained on the resulting descriptors. The original Matlab implementation of TS was rewritten in Python, using the standard linear algebra libraries [41] and numba [42] to accelerate loops where possible.

As done in [15], we use $C_{\text{svm}} = 1$ to train the linear SVMs in all experiments.

4.2. Datasets used in the experiments

For our experiments, we use three well known fine-grained image categorization datasets, all of which include pre-defined train/test splits:

- Caltech UCSD Birds-200-2011 [1] (CUB). Animal species recognition dataset with 200 bird species, which extends the earlier CUB-200 dataset by increasing the number of images per class. The dataset contains a total of 11,788 images, with a standard split of 5994 images for training and 5794 for testing. The number of images per class ranges from 41 to 60. Part annotations and bounding boxes are provided for all the images.
- Stanford Cars Dataset (CARS) [4] Car model recognition dataset with 196 categories. Classes include the model and year of the car, for example “2012 Tesla Model S” or “2012 BMW M3”. The dataset contains a total of 16,185 images, with a standard split of 8144 images for training+validation and 8041 for testing. Bounding boxes are provided for all the images.
- 102 Category Flower Dataset [3] (Flowers). Plant species recognition dataset with 102 flower species commonly occurring in the United Kingdom. The dataset contains a total of 8189 images, with a standard split of 2040 images for training+validation and 6149 for testing. The number of images per class ranges from 40 to 258. Segmentation data is provided for the images.

Training and test images were preprocessed as follows. First, bounding boxes were used for CUB and CAR datasets to extract the relevant region of the images. In the case of the Flower dataset, bounding boxes are not explicitly provided, so the entire images were kept. Secondly, the resulting images were padded with zeros to make them square, and resized to the appropriate size depending on the CNN used in the experiment.[9] Finally, color preprocessing was applied as required.[10]

4.3. Classification accuracy and inference-time

Tables 5 and 6 compare the accuracies and inference-times achieved by the different approaches described in Section 4.1 using SqueezeNet and GoogLeNet, respectively. To compensate for the stochastic nature of some of the methods evaluated, each experiment was executed ten times. The average accuracy is displayed together with the standard deviation. Regarding inference-time results, times are reported in the format $T_1/T_2/T_3$, where T_1 represents the time required for the image to be passed through the CNN, T_2 is the time needed to generate the final descriptor (either by full bilinear pooling or the corresponding compact bilinear pooling method), and T_3 is the time taken by the final linear classifier to emit a prediction. Note that unlike T_1 and T_2, T_3 is affected by the number of classes in the dataset. The timings reported in

[9] By default, SqueezeNet and GoogLeNet have input sizes of 227 × 227 and 224 × 224 respectively.

[10] The GoogLeNet implementation used requires pixel values in the range [−1,1]. SqueezeNet requires conversion from RGB to BGR and color zero-centering with respect to the ImageNet dataset.

Table 5
Comparison of compact bilinear pooling methods using SqueezeNet v1.1 chopped at *fire9* [36] as the base network. Inference time results are for the CUB dataset (i.e., 200 categories).

Method	Descript. size (k)	Acc. (%) CUB [1]	Acc. (%) CARS [4]	Acc. (%) Flowers [3]	Time (ms) Pi 3 Model B+	Time (ms) Pi Zero W
Baseline	$(13 \times 13 \times 512)$	46.46	49.07	71.83	156/0/119 Total: 273	1989/0/490 Total: 2,481
FB	512^2	66.05	63.42	83.34	149/22/360 Total: 539	1996/1042/1493 Total: 4,540
CBP-KRP	2000	60.78 ± 0.29	54.36 ± 0.39	80.75 ± 0.19	156/105/2 Total: 265	1962/490/11 Total: 2,461
TS	2000	60.17 ± 0.22	54.13 ± 0.24	80.60 ± 0.30	154/340/2 Total: 500	1968/1162/11 Total: 3,152
RM	2000	59.51 ± 0.28	53.12 ± 0.35	79.48 ± 0.29	154/483/2 Total: 644	1950/1865/11 Total: 3,828
CBP-KRP	3500	62.26 ± 0.24	57.28 ± 0.36	81.63 ± 0.20	155/113/4 Total: 276	1966/582/20 Total: 2,573
TS	3500	61.80 ± 0.25	57.35 ± 0.30	81.48 ± 0.24	148/747/4 Total: 908	1950/2920/20 Total: 4,895
RM	3500	60.63 ± 0.25	55.84 ± 0.31	80.17 ± 0.27	147/859/4 Total: 1,021	1967/3268/20 Total: 5,257
CBP-KRP	5000	62.94 ± 0.16	58.68 ± 0.42	82.04 ± 0.29	155/123/6 Total: 287	1965/697/28 Total: 2,690
TS	5000	62.85 ± 0.29	58.84 ± 0.26	81.95 ± 0.20	152/916/6 Total: 1,077	1960/3791/28 Total: 5,797
RM	5000	61.11 ± 0.17	56.97 ± 0.25	80.49 ± 0.19	152/1224/6 Total: 1,388	1951/4668/28 Total: 6,665

Table 6
Comparison of compact bilinear pooling methods using GoogLeNet chopped at *inception (4e)* [36] as the base network. Inference time results are for the CUB dataset (i.e., 200 categories).

Method	Descript. size (k)	Acc. (%) CUB [1]	Acc. (%) CARS [4]	Acc. (%) Flowers [3]	Time (ms) Pi 3 Model B+	Time (ms) Pi Zero W
Baseline	$(14 \times 14 \times 832)$	47.03	56.05	77.49	542/0/223 Total: 770	11629/0/968 Total: 12,684
FB	832^2	74.83	75.46	89.78	545/74/951 Total: 1,571	11499/3083/49374 Total: 100,280
CBP-KRP	2000	68.68 ± 0.27	62.88 ± 0.32	88.28 ± 0.23	537/156/2 Total: 704	11440/740/12 Total: 12,174
TS	2000	67.44 ± 0.22	61.31 ± 0.26	87.90 ± 0.21	534/396/2 Total: 944	11770/1437/12 Total: 13,155
RM	2000	67.56 ± 0.33	62.17 ± 0.34	87.82 ± 0.21	539/820/2 Total: 1,359	11358/3417/12 Total: 14,781
CBP-KRP	3500	70.14 ± 0.45	65.46 ± 0.34	89.02 ± 0.24	537/171/4 Total: 712	11627/862/20 Total: 12,553
TS	3500	69.61 ± 0.35	64.20 ± 0.24	88.73 ± 0.23	536/872/4 Total: 1,426	11478/3636/20 Total: 15,142
RM	3500	69.20 ± 0.27	64.57 ± 0.22	88.39 ± 0.20	532/1477/4 Total: 2,020	11514/6084/20 Total: 17,614
CBP-KRP	5000	71.04 ± 0.22	66.84 ± 0.32	89.24 ± 0.15	546/182/7 Total: 731	11481/985/30 Total: 12,504
TS	5000	70.52 ± 0.27	65.68 ± 0.35	89.05 ± 0.17	526/1074/6 Total: 1,619	11452/4867/29 Total: 16,363
RM	5000	69.87 ± 0.30	65.71 ± 0.28	88.56 ± 0.17	539/2100/6 Total: 2,651	11554/8682/29 Total: 20,243

the tables are for a training dataset with 200 categories (e.g., the CUB dataset). Total inference times are also reported.[11]

Looking at the accuracies in Tables 5 and 6, we can see that CBP-KRP outperformed the alternative compact bilinear pooling methods is most cases, providing the closest approximation to the accuracy of full bilinear pooling. Notably, this is achieved while maintaining much lower total inference times. For instance, using SqueezeNet and $k = 5,000$ on the Raspberry Pi 3 Model B+, the total inference time with CBP-KRP as the compact bilinear pooling method is 287 ms, while with TS and RM inference times break the one second mark. In addition, using CBP-KRP also results in lower inference times when compared with the full bilinear approach. In fact, CBP-KRP provided inference times roughly two times lower than those of full bilinear pooling on the Pi 3 Model B+, and up to eight times lower on the Pi Zero W. This efficiency is in part achieved thanks to the sparse nature of the vectors used in CBP-KRP, which enables using fast sparse matrix multiplication routines for the projection. This supports our claim that, when considering low computational power devices, compact bilinear pooling methods can be useful not only to reduce models' memory requirements but to achieve lower inference times.

Another important aspect to consider when analyzing these results is the final model size achieved when using the different methods. As mentioned before, Table 1 shows some useful figures in this respect. Both CNNs used have a relatively low initial model size with 4.8 MB for SqueezeNet and 25.7 MB for GoogLeNet. In our experimental setup, using full bilinear pooling increases these model sizes by 200 and 528 MB respectively, as a consequence of the high number of parameters of a linear classifier trained on 512^2 or 832^2 features, with 200 classes and a one-vs-all scheme. This of course is a problem if we want our models to run on devices with as little as 512 MB of RAM, which might also have other running processes competing for resources. Model size is also a problem when using compact bilinear pooling via RM, as the parameters needed by RM itself can require and important amount of memory. For instance, when using SqueezeNet, RM required 19.5 MB of additional memory, making the final model five times as heavy as the base CNN. Conversely, compact bilinear pooling via TS and CBP-KRP have a low memory footprint. As an example, consider the case were we use GoogLeNet. With TS, only 13 KB of additional memory are required to store its parameters. With CBP-KRP, 406 KB are required for the same purpose. This difference in the memory requirements of TS and CBP-KRP is significant, but has a limited impact in the final model size given the 25 MBs of the base network and the 3.8 MBs of the linear classifier, which do not vary depending on whether TS or CBP-KRP are used.

4.4. Results with fine-tuning

As explained in Section 3.3, one interesting feature of the proposed algorithm is its compatibility with the back-propagation algorithm, which makes it possible to include it as an intermediate layer of end-to-end trainable models. In our experiments so far, we have focused on a simple transfer learning use case where only the final layer of the model is trained (i.e., the linear SVM), while the weights of the remaining layers are fixed. However, it is also common, if enough training data is available, to fine-tune the weights of the entire model by running some iterations of gradient descent with a low learning rate. Conveniently, the inference-time and size of the model do not change with this process. Therefore, models can be fine-tuned on computers with specialized hardware and then deployed in low power devices, obtaining a potential boost in accuracy with no increase in inference times. In this section, we show that CBP-KRP is compatible with this fine-tuning approach and how it can improve the performance with respect to transfer learning without fine-tuning.

[11] Small discrepancies exist between total inference times reported and the sum of T_1, T_2 and T_3. This is because total inference times were measured independently and not computed as $T_1 + T_2 + T_3$. All the timings reported correspond to the lowest execution time among ten runs.

Table 7
Accuracies obtained using CBP-KRP with SqueezeNet and GoogLeNet, fine-tuning all layers of the pre-trained base network on the target dataset. Reported accuracies are the average over five runs, together with the average improvement with respect to the same experiment without fine-tuning.

Method	Base Network	Descript. size (k)	Acc. (%) CUB [1]	Acc. (%) CARS [4]	Acc. (%) Flowers [3]
CBP-KRPfine-tuned	SqueezeNet v1.1at *fire9*	2000	68.50 ± 0.13(7.72 ↑)	66.50 ± 0.18(12.14 ↑)	84.99 ± 0.27(4.24 ↑)
CBP-KRPfine-tuned	SqueezeNet v1.1at *fire9*	3500	69.53 ± 0.42(7.24 ↑)	68.59 ± 0.12(11.31 ↑)	85.47 ± 0.10(3.84 ↑)
CBP-KRPfine-tuned	SqueezeNet v1.1at *fire9*	5000	69.92 ± 0.20(6.98 ↑)	69.66 ± 0.11(10.98 ↑)	85.65 ± 0.08(3.61 ↑)
CBP-KRPfine-tuned	GoogLeNetat *inception (4e)*	2000	80.13 ± 0.51(11.45 ↑)	82.08 ± 0.35(19.20 ↑)	92.61 ± 0.16(4.33 ↑)
CBP-KRPfine-tuned	GoogLeNetat *inception (4e)*	3500	80.71 ± 0.06(10.57 ↑)	83.21 ± 0.22(17.75 ↑)	92.72 ± 0.20(3.70 ↑)
CBP-KRPfine-tuned	GoogLeNetat *inception (4e)*	5000	81.11 ± 0.10(10.07 ↑)	83.79 ± 0.20(16.95 ↑)	92.94 ± 0.10(3.70 ↑)

We adopt a two step fine-tuning procedure similar to the one used in [15]: The process begins by chopping the pre-trained CNN model, keeping the layers before the selected cutoff point. After this, CBP-KRP is initialized and appended to the CNN as a layer in the model. Then, a softmax layer is added as the final layer of the model. The first step in the training procedure consist in training this softmax layer alone, without altering the rest of the weights of the model. Then, with the model assembled and the final layer already trained, all the weights in the model are fine-tuned by executing a number of iterations of gradient descent. As explained in Section 3.3, the parameters of CBP-KRP are excluded from this fine-tuning process in order to preserve their sparsity. For our experiments, we used Adam [43] as the optimizer, and set the learning rate to 0.001 with a learning rate decay of 0.1. Batch size was set to 32 and the number of epochs to 20.

Table 7 shows the accuracies resulting from applying this approach with SqueezeNet and GoogLeNet as the base CNN, and different output dimensions for CBP-KRP. As we can see, accuracies improved in all cases as a result of fine-tuning. The improvements in the accuracy ranged from 3.61 to 19.20, with the higher improvements occurring for the GoogLeNet CNN. These results evidence the potential of fine-tuning models which include compact bilinear pooling as an intermediate layer, and the compatibility of CBP-KRP with this approach.

4.5. Hyperparameter selection

One possible drawback of the proposed method is that the end-user must specify the value of a number of hyperparameters, which can be challenging when the underlying effects of these hyperparameters are not known. This subsection tries to mitigate this problem by providing a detailed description of the different hyperparameters of CBP-KRP, and exploring the effect of modifying each of them.

Looking at Algorithm 1, we can see that CBP-KRP has four hyperparameters whose values must be provided. These are the total number of random vectors generated (p), their sparsity level (s), the number of vectors summed to form each projection vector in the feature space (t), and the desired output dimension (k).

The hyperparameter p, which controls the number of unique random vectors generated by the algorithm, was introduced in [20] to reduce the computational cost of the kernelized Random Projection. As explained in Section 3.1, instead of using $2t$ distinct vectors for each output component, our algorithm generates a collection with p vectors, and reuses some of them in order to reduce costs. Therefore, p must be set to be $2tk \geq p > 2t$. Larger values of p reduce the re-usage of vectors, improving performance at the cost of longer running times. If p is set to $2tk$, no vector repetition will occur at all. Similarly, lower values of p sacrifice some performance to achieve a faster execution. Therefore, this hyperparameter can be used to control the performance/efficiency trade-off, without modifying the dimension of the output representation, which may have further implications. Fig. 3 illustrates the effect in accuracy and execution times of using different values for p. Conveniently, we can see that the accuracy grows rapidly with p, while as explained in Section 3.2 embedding times with CBP-KRP are linear in p.

For is part, k is a common hyperparameter in most kernel approximation methods which controls the number of features generated to approximate the kernel. Therefore, the hyperparameter k also defines a trade-off between accuracy and efficiency. The main difference is that, as opposed to p, k determines the dimensionality of the resulting descriptors, which might have implications for subsequent steps in the processing chain (e.g., for the final linear classifier in our case). Again, Fig. 3 explores this trade-off, showing that the accuracy grows quickly as k increases.

The hyperparameter t determines the number of random vectors summed to form the projection vectors in the feature space. As explained in Section 3, forming the projection vectors as the sum of t random vectors results in a reduced dependence among their entries, which as shown in [20] is key for the distance-preservation properties of Random Projection. However, the same study revealed that the effect of t in classification accuracies is limited, and recommended using small values of this hyperparameter when the generated representations are intended for classification.

Finally, hyperparameter s determines the degree of sparsity of the generated random vectors. In particular, the entries of these vectors are zero with probability $1 - 1/s$. Therefore, using a relatively large s enables us to reduce computational and storage costs. Furthermore, using projection vectors with a certain degree of sparsity does not necessary have a negative impact in the classification accuracy, as sparse Random Projections are known to

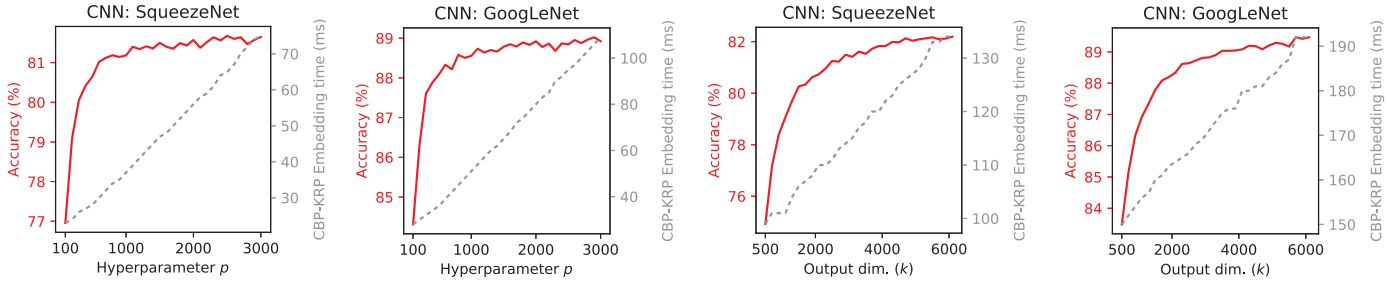

Fig. 3. Effect of using different values for the hyperparameter p and the output dimension k. Experiments are for the Flowers dataset. When exploring different values of p, k was fixed to 3500. Similarly, p was fixed to 5000 when exploring the effect of k. Embedding times are for the Raspberry Pi 3 Model B+.

Table 8
Accuracies obtained by CBP-KRP on the three datasets studied, with SqueezeNet and GoogLeNet as the base network and using different values for hyperparameters t and s. Hyperparameters p and k were fixed to 5000 and 2000 respectively. The best result for each dataset and base network is stressed in bold.

CBP-KRP	Accuracy with SqueezeNet (%)			Accuracy with GoogLeNet (%)		
Hyperparameters	CUB [1]	CARS [4]	Flowers [3]	CUB [1]	CARS [4]	Flowers [3]
$t = 2$, $s = 50$	60.22 ± 0.26	54.15 ± 0.36	80.55 ± 0.30	68.07 ± 0.27	62.26 ± 0.37	88.13 ± 0.21
$t = 4$, $s = 50$	59.86 ± 0.38	53.78 ± 0.37	80.22 ± 0.24	67.58 ± 0.32	61.72 ± 0.44	87.88 ± 0.21
$t = 6$, $s = 50$	59.66 ± 0.32	53.42 ± 0.36	80.00 ± 0.26	67.37 ± 0.33	61.15 ± 0.40	87.73 ± 0.21
$t = 2$, $s = 100$	**60.75** ± 0.35	**54.30** ± 0.39	**80.73** ± 0.26	68.73 ± 0.36	62.88 ± 0.43	88.25 ± 0.20
$t = 4$, $s = 100$	60.39 ± 0.35	54.16 ± 0.39	80.57 ± 0.28	68.29 ± 0.34	62.31 ± 0.35	88.19 ± 0.19
$t = 6$, $s = 100$	60.15 ± 0.40	54.06 ± 0.35	80.49 ± 0.27	67.88 ± 0.33	61.74 ± 0.44	88.06 ± 0.26
$t = 2$, $s = 200$	60.41 ± 0.38	52.65 ± 0.46	80.25 ± 0.34	68.72 ± 0.39	**63.21** ± 0.48	88.15 ± 0.28
$t = 4$, $s = 200$	60.74 ± 0.33	53.86 ± 0.43	80.60 ± 0.39	**68.77** ± 0.41	62.86 ± 0.37	**88.30** ± 0.31
$t = 6$, $s = 200$	60.63 ± 0.44	54.24 ± 0.36	80.70 ± 0.21	68.55 ± 0.42	62.46 ± 0.36	88.17 ± 0.22

perform well in practice [34]. Moreover, sparsity has been shown to be a powerful tool in the context of deep learning, as it can contribute to mitigate over-fitting [44].

It must be noted, however, that since the projection vectors in the kernel feature space are built as the sum of t vectors, the sparsity level of the final projection vectors will also be affected by t, and not only by s. Hence, t and s should be jointly selected. Table 8 shows the accuracies obtained by CBP-KRP on the three datasets studied, using different values for hyperparameters t and s. Luckily, the results suggest that the proposed method is fairly robust to the selection of these hyperparameters. Particularly, the combination used in the comparisons of the previous section, $t = 2$ and $s = 100$, resulted in either the best or the second best result in all experiments. In some cases, a slight improvement in the accuracy was achieved when increasing the sparsity by setting $s = 200$ and using $t = 4$ or $t = 6$.

5. Conclusions

This paper builds upon the ideas of [18,20] to propose CBP-KRP, a novel method to create compact feature descriptors which capture most of the power of full bilinear pooling descriptors [15]. Following the insights provided by [18], we proposed an efficient method to approximate a Random Projection of the full bilinear descriptor, mostly preserving its discriminative information while greatly reducing the dimension of the final descriptor. This was achieved by adapting the ideas from [20], and exploiting the close relation between the bilinear pooling operation and homogeneous polynomial kernels. We also derived back-propagation for the proposed algorithm, showing that it can be used as a building block in end-to-end trainable models.

Our experimental results show that, for three common fine-grained image categorization datasets, our method produces the best approximation to the accuracy of full bilinear pooling, outperforming existing compact bilinear pooling methods. Moreover, this is achieved while running significantly faster than TS and RM-based compact bilinear pooling on low computational power devices such as those from the Raspberry Pi ecosystem, and also faster than full bilinear pooling. In addition, the number of parameters used by our algorithm is relatively low, solving the memory issues that emerge when using full bilinear descriptors. As a consequence, our algorithm could be useful in embedded systems or other low computational power scenarios where tight computation and memory constraints exist.

Following previous studies on the topic of compact bilinear pooling, we focused on the case where a single CNN is used to form the bilinear descriptors [18]. However, an interesting line for future work would be the possibility of extending CBP-KRP to the case where bilinear descriptors are formed as the outer product of the descriptors extracted by two different CNNs, as this could have applications in multi-modal problems [45,46]. In addition, we would like to explore the applicability of our algorithm in areas such as Internet of Things, Wearable technology or Embedded Systems [47], where efficient fine-grained image understanding methods could be of great use.

Declaration of Competing Interest

There are no conflicts of interest to declare.

Acknowledgement

The research of Daniel López-Sánchez has been financed by the Ministry of Education, Culture and Sports of the Spanish Government, University Faculty Training (FPU) programme, reference number FPU15/02339.

References

[1] C. Wah, S. Branson, P. Welinder, P. Perona, S. Belongie, The Caltech-UCSD Birds-200-2011 Dataset, Technical Report, California Institute of Technology, 2011.
[2] A. Khosla, N. Jayadevaprakash, B. Yao, L. Fei-Fei, Novel dataset for fine-grained image categorization, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, First Workshop on Fine-Grained Visual Categorization, Colorado Springs, CO, 2011.
[3] M.-E. Nilsback, A. Zisserman, Automated flower classification over a large number of classes, in: Proceedings of the Indian Conference on Computer Vision, Graphics and Image Processing, 2008.
[4] J. Krause, M. Stark, J. Deng, L. Fei-Fei, 3D object representations for fine-grained categorization, in: Proceedings of the Fourth International IEEE Workshop on 3D Representation and Recognition (3DRR), Sydney, Australia, 2013.
[5] S. Maji, J. Kannala, E. Rahtu, M. Blaschko, A. Vedaldi, Fine-Grained Visual Classification of Aircraft, Technical Report, 2013.
[6] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, A. Oliva, Learning deep features for scene recognition using places database, in: Proceedings of the Advances in Neural Information Processing Systems, 2014, pp. 487–495.
[7] S. Branson, G. Van Horn, S. Belongie, P. Perona, Bird species categorization using pose normalized deep convolutional nets, arXiv:1406.2952 (2014).
[8] Z. Xu, S. Huang, Y. Zhang, D. Tao, Augmenting strong supervision using web data for fine-grained categorization, in: Proceedings of the IEEE International Conference on Computer Vision, 2015, pp. 2524–2532.
[9] J. Krause, B. Sapp, A. Howard, H. Zhou, A. Toshev, T. Duerig, J. Philbin, L. Fei-Fei, The unreasonable effectiveness of noisy data for fine-grained recognition, in: Proceedings of the European Conference on Computer Vision, Springer, 2016, pp. 301–320.
[10] A.R. Chowdhury, T.-Y. Lin, S. Maji, E. Learned-Miller, One-to-many face recognition with bilinear CNNS, in: Proceedings of the IEEE Winter Conference on Applications of Computer Vision (WACV), IEEE, 2016, pp. 1–9.
[11] C. Feichtenhofer, A. Pinz, A. Zisserman, Convolutional two-stream network fusion for video action recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 1933–1941.
[12] E. Ustinova, Y. Ganin, V. Lempitsky, Multi-region bilinear convolutional neural networks for person re-identification, in: Proceedings of the 14th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS), IEEE, 2017, pp. 1–6.
[13] A. Alzu'bi, A. Amira, N. Ramzan, Content-based image retrieval with compact deep convolutional features, Neurocomputing 249 (2017) 95–105.
[14] Q. Sun, Q. Wang, J. Zhang, P. Li, Hyperlayer bilinear pooling with application to fine-grained categorization and image retrieval, Neurocomputing 282 (2018) 174–183.

[15] T.-Y. Lin, A. RoyChowdhury, S. Maji, Bilinear CNN models for fine-grained visual recognition, in: Proceedings of the IEEE International Conference on Computer Vision, 2015, pp. 1449–1457.
[16] W. Shi, J. Cao, Q. Zhang, Y. Li, L. Xu, Edge computing: vision and challenges, IEEE Internet Things J. 3 (5) (2016) 637–646.
[17] H. Huang, H. Zhou, X. Yang, L. Zhang, L. Qi, A.-Y. Zang, Faster r-CNN for marine organisms detection and recognition using data augmentation, Neurocomputing 337 (2019) 372–384.
[18] Y. Gao, O. Beijbom, N. Zhang, T. Darrell, Compact bilinear pooling, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 317–326.
[19] D. Achlioptas, Database-friendly random projections: Johnson-Lindenstrauss with binary coins, J. Comput. Syst. Sci. 66 (4) (2003) 671–687.
[20] D. López-Sánchez, A.G. Arrieta, J.M. Corchado, Data-independent random projections from the feature-space of the homogeneous polynomial kernel, Pattern Recognit. 82 (2018) 130–146.
[21] J.B. Tenenbaum, W.T. Freeman, Separating style and content with bilinear models, Neural Comput. 12 (6) (2000) 1247–1283.
[22] S. Taheri, O. Toygar, On the use of DAG-CNN architecture for age estimation with multi-stage features fusion, Neurocomputing 329 (2019) 300–310.
[23] P. Kar, H. Karnick, Random feature maps for dot product kernels, in: Proceedings of the 15th International Conference on Artificial Intelligence and Statistics (AISTATS), 2012, pp. 583–591.
[24] N. Pham, R. Pagh, Fast and scalable polynomial kernels via explicit feature maps, in: Proceedings of the Nineteenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2013, pp. 239–247.
[25] M. Charikar, K. Chen, M. Farach-Colton, Finding frequent items in data streams, in: Proceedings of the International Colloquium on Automata, Languages, and Programming, Springer, 2002, pp. 693–703.
[26] J.-H. Kim, K.-W. On, W. Lim, J. Kim, J.-W. Ha, B.-T. Zhang, Hadamard product for low-rank bilinear pooling, arXiv:1610.04325 (2016).
[27] Z. Yu, J. Yu, C. Xiang, J. Fan, D. Tao, Beyond bilinear: generalized multimodal factorized high-order pooling for visual question answering, IEEE Trans. Neural Netw. Learn. Syst. 99 (2018) 1–13.
[28] S. Dasgupta, A. Gupta, An elementary proof of a theorem of Johnson and Lindenstrauss, Random Struct. Algorithms 22 (1) (2003) 60–65.
[29] K. Zhao, A. Alavi, A. Wiliem, B.C. Lovell, Efficient clustering on Riemannian manifolds: a kernelised random projection approach, Pattern Recognit. 51 (2016) 333–345.
[30] A. Alavi, A. Wiliem, K. Zhao, B.C. Lovell, C. Sanderson, Random projections on manifolds of symmetric positive definite matrices for image classification, arXiv:1403.0700 (2014).
[31] D. López-Sánchez, J.M. Corchado, A.G. Arrieta, Data-independent random projections from the feature-map of the homogeneous polynomial kernel of degree two, Inf. Sci. 436 (2018) 214–226.
[32] P. Indyk, R. Motwani, Approximate nearest neighbors: towards removing the curse of dimensionality, in: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, ACM, 1998, pp. 604–613.
[33] R.I. Arriaga, S. Vempala, An algorithmic theory of learning: robust concepts and random projection, in: Proceedings of the Fortieth Annual Symposium on Foundations of Computer Science, IEEE, 1999, pp. 616–623.
[34] P. Li, T.J. Hastie, K.W. Church, Very sparse random projections, in: Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2006, pp. 287–296.
[35] L. Breiman, Probability, Society for Industrial and Applied Mathematics (SIAM), 1992, pp. 237–238.
[36] F.N. Iandola, S. Han, M.W. Moskewicz, K. Ashraf, W.J. Dally, K. Keutzer, SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5 mb model size, arXiv:1602.07360 (2016).
[37] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 1–9.
[38] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D.G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng, TensorFlow: a system for large-scale machine learning, in: Proceedings of the Twelfth USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2016, pp. 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.
[39] F. Chollet, et al., Keras, 2015, (https://keras.io).
[40] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, C.-J. Lin, LIBLINEAR: a library for large linear classification, J. Mach. Learn. Res. 9 (2008) 1871–1874.
[41] T.E. Oliphant, A Guide to NumPy, 1, Trelgol Publishing USA, 2006.
[42] S.K. Lam, A. Pitrou, S. Seibert, Numba: a LLVM-based python JIT compiler, in: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, ACM, 2015, p. 7.
[43] D.P. Kingma, J. Ba, Adam: a method for stochastic optimization, arXiv:1412.6980 (2014).
[44] Q. Xu, M. Zhang, Z. Gu, G. Pan, Overfitting remedy by sparsifying regularization on fully-connected layers of CNNs, Neurocomputing 328 (2019) 69–74.
[45] A. Fukui, D.H. Park, D. Yang, A. Rohrbach, T. Darrell, M. Rohrbach, Multimodal compact bilinear pooling for visual question answering and visual grounding, arXiv:1606.01847 (2016).
[46] C. Hong, J. Yu, J. Wan, D. Tao, M. Wang, Multimodal deep autoencoder for human pose recovery, IEEE Trans. Image Process. 24 (12) (2015) 5659–5670.
[47] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, M. Ayyash, Internet of things: a survey on enabling technologies, protocols, and applications, IEEE Commun. Surv. Tutor. 17 (4) (2015) 2347–2376.

Daniel L ópez-S ánchez (M.Sc.) obtained a Computer Science degree (highest GPA award) in 2015 and an M.Sc. in Artificial Intelligence in 2016 at the University of Salamanca (Spain). In 2016, he received the FPU grant from the Spanish Government and started pursuing a Ph.D. at the BISITE Research Group. His research interest focus in the field of machine learning, especially in the subfields of dimensionality reduction and kernel methods. He is also interested in developing novel applications of the Deep Learning paradigm. He has participated as a co-author in papers published in recognized international conferences and journals.

Angélica Gonz ález-Arrieta (Ph.D.) Received a Ph.D. in Computer Science from the University of Salamanca in 2000. She is currently a Lecturer in Salamancas University Department of Computer Science and has attended several Master's courses. She is further a professor and tutor for UNED (Universidad Española de Educaci ón a Distancia, Spain's Open University). In the past, she carried out relevant administrative tasks, such as Academic Secretary of the Science Faculty (1996–2000) and Chief of Staff for the University of Salamanca (2000–2003). From 1990, she has cooperated with the Home Ministry, and from 2008 with the Home and Justice Counsel of the local government (Junta de Castilla y Le ón). She is a member of the research group BISITE (http://bisite.usal.es) and has lead several research projects sponsored by both public and private institutions in Spain. She is the coauthor of several works published in magazines, workshops, meetings, and symposia.

Juan M. Corchado (Ph.D.) Received a Ph.D. in Computer Science from the University of Salamanca in 1998 and a Ph.D. in Artificial Intelligence (AI) from the University of Paisley, Glasgow (UK) in 2000. At present, he is Vice President for Research and Technology Transfer since December 2013 and a Full Professor with Chair at the University of Salamanca. He is the Director of the Science Park of the University of Salamanca and Director of the Doctoral School of the University. He has been elected twice as the Dean of the Faculty of Science at the University of Salamanca. He has led several Artificial Intelligence research projects sponsored by Spanish and European public and private sector institutions and has supervised seven Ph.D. students. He is the coauthor of over 230 books, book chapters, journal papers, technical reports, etc. Since January 2015, He is Visiting Professor at Osaka Institute of Technology. He is also member of the Advisory group on Online Terrorist Propaganda of the European Counter Terrorism Centre (EUROPOL). He is also editor and Editor-in-Chief of Specialized Journals like ADCAIJ (Advances in Distributed Computing and Artificial Intelligence Journal), IJDCA (International Journal of Digital Contents and Applications) and OJCST (Oriental Journal of Computer Science and Technology).