Expressing Neural Network layers as Fast-Transforms

Luc Giffon
QARMA team - LIS
Aix-Marseille Université
Marseille, France

September 20, 2019

Abstract

We apply the `palm4MSA` algorithm on a pretrained neural network in order to express the fully-connected weights matrices and the matrix of convolutional filters as fast-transforms. This allows a drastic reduction of the number of parameters in the network and the associated speed-up in computation at inference time when deployed on a single-threaded architecture. This post-training modification of neural networks doesn't impair accuracy performance (or we hope so).

Motivation: No GPU needed at inference time? Less RAM needed? Lower carbon footprint?

1 Introduction

It is widely accepted that neural networks have been a game changer in the race for best classification accuracy on many image classification and natural language processing (NLP) benchmarks. These, not so new anymore, families of models are able to take advantage of the parallelization power of modern GPUs to learn from udge databases and tune millions of parameters to achieve a given task.

It must be emphasized though that these amazing results come at a price, both economical and ecological. On one hand, recent work [7] have pointed out that training some NLP models have a significant energy consumption cost that is directly translatable to financial or carbon footprint considerations. On the other hand, the vast majority of power consumption for neural networks is spent for inference and not for training[4] [1]. Even though computer vision models aren't as demanding in energy than NLP does yet, it is clear that deploying them at large scale, as they are about to be, will bring concerns of the same nature.

These energy consumption concerns are directly related to the number of parameters necessary to achieve such excellent accuracy results. Indeed, it mainly comes from the many multiplications involved in the inner linear transformations at each layer of neural networks. This udge number of parameter brings also the problem of storing some models in GPUs VRAM, making them particularly unhandy to deploy on small devices such as smartphones. Additionaly, such devices doesn't even have embedded GPU, in which case the amount of computation becomes completely untractable for simple CPU hardware.

Introduction to be continued.... mais vous avez saisi l'idée

Speak about:

- We leverage recent advances in optimisation to express layers as factorization of sparse matrices that mimics fast-transform computation and reduce the nbr of parameters in the network + reduce computation time from a single-threaded perspective

2 Related work

Speak about:

- DeepfriedConvnet: leverage fast hadamard transform but induces structural bias on weight matrices + not implemented for convolution layers
- Tensor decomposition: see references in the survey [3]
- Pruning of weights: see references in the survey [3]

3 Linear transformations in neural networks

To keep the notations clear, we chose to not use indexes to refer to the position of layers in the following explanations; we assume without loss of generality that the formulas apply to any layer w.r.t. their own dimensions and parameters.

A part of the computation of any layer in a neural network, either fully-connected or convolutional, consist in some linear transformation. In this section, we describe these linear transformations in both cases.

In a fully-connected layer The output $\mathbf{z} \in \mathbb{R}^D$ of a fully-connected layer is given by:

$$\mathbf{z} = \sigma(\mathbf{W}\mathbf{x}) \tag{1}$$

, where σ is some non-linear activation function, $\mathbf{W} \in \mathbb{R}^{D \times d}$ is the weight matrix of the layer and $\mathbf{x} \in \mathbb{R}^d$ is the output of the preceding layer. The fully-connected layer computes first a linear transformation of its input in order to compute the final, non-linear, transformation.

In a convolutional layer Let us define two reshape operators:

First, $\mathcal{R}_{h,w}$ parameterized by an height h and a width w that takes a tensor of shape $(H \times W \times C)$ as input:

$$\mathcal{R}_{h,w} : \mathbb{R}^{H \times W \times C} \mapsto \mathbb{R}^{HW \times Chw} \tag{2}$$

. This reshape operation creates the matrix of all vectorized patches of height h and width w. We again preserve simplicity in notation here, assuming without loss of generality that the stride used by $\mathcal{R}_{h,w}$ is equal to 1 (e.g. one patch at each 2-D coordinate of the input tensor) and that the input tensor is padded with $\lfloor \frac{h}{2} \rfloor$ zeros verticaly and $\lfloor \frac{w}{2} \rfloor$ zeros horizontaly (e.g. we do not need to worry about what happens on the edges of the input tensor);

Second, $\mathcal{T}_{H,W}$ parameterized by the height H and width W of the reconstructed tensor:

$$\mathcal{T}_{H,W} : \mathbb{R}^{HW \times K} \mapsto \mathbb{R}^{H \times W \times K} \tag{3}$$

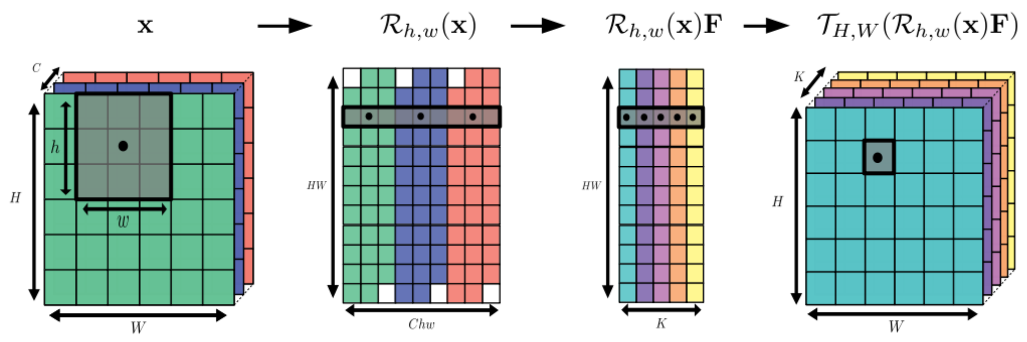
$\mathbf{x}$
$\mathcal{R}_{h,w}(\mathbf{x})$
$\mathcal{R}_{h,w}(\mathbf{x})\mathbf{F}$
$\mathcal{T}_{H,W}(\mathcal{R}_{h,w}(\mathbf{x})\mathbf{F})$
C
h
H
w
W
HW
Chw
HW
K
K
H
W

Figure 1: Depiction of the reshape operations and localy linear transformations computed in the convolutional layers. We've tried to represent the evolution of the input tensor as it goes along the convolution processing. For each drawing, the mathematical value of what it is supposed to represent is written on top of it. The scale isn't respected between steps but we've tried to bring consistency in the channels depiction. The greyed box represent the receptive field of the convolution filters at one particular coordinate. This coordinate is depicted by the black dot and can be followed along operations. The different colors represent the different channels either in the input (C channels) or the output (K channels) tensor. The whited squares in the second step recall the zero padding of the input.

. This reshape operation is an inverse flatten operation on an image with K channels. Figure 1 gives a visual representation of these reshape operations.

With these reshape operators, the output $\mathbf{Z} \in \mathbb{R}^{H \times W \times K}$ of a convolutional layer with K filters is given by:

$$\mathbf{Z} = \sigma(\mathcal{T}_{H,W}(\mathcal{R}_{h,w}(\mathbf{X})\ \mathbf{F})) \tag{4}$$

, where σ is some non-linear activation function, $\mathbf{F} \in \mathbb{R}^{Chw \times K}$ is the matrix of weights describing the convolution filters and $\mathbf{X} \in \mathbb{R}^{H \times W \times C}$ is the output 3-D tensor of the preceding layer. The convolutional layer uses the matrix of filters $\mathbf{F}$ in order to compute localy linear transformations of its input patches defined by the reshape operator $\mathcal{R}_{h,w}$.

4 Learning fast-transform Structures

This whole section "Learning fast-transform structures" has been copy-pasted from the quick-means paper. Must modify it.

Linear operators structured as products of sparse matrices. The popularity of some linear operators from $\mathbb{R}^D$ to $\mathbb{R}^D$ (with $D < \infty$) like Fourier or Hadamard transforms comes from both their mathematical properties and their ability to compute the mapping of some input $\mathbf{x} \in \mathbb{R}^D$ with efficiency, typically in $\mathcal{O}\left(D \log D\right)$ in lieu of $\mathcal{O}\left(D^2\right)$ operations. The core feature of the related fast algorithms is that the matrix $\mathbf{W} \in \mathbb{R}^{D \times D}$ of such linear operators can be written as the product $\mathbf{W} = \Pi_{q \in [\![Q]\!]} \mathbf{S}_q$ of $Q = \mathcal{O}\left(\log D\right)$ sparse matrices $\mathbf{S}_q$ with $\left\|\mathbf{S}_q\right\|_0 = \mathcal{O}\left(D\right)$ non-zero coefficients per factor [5, 6]: for any vector $\mathbf{x} \in \mathbb{R}^M$, $\mathbf{Wx}$ can thus be computed as $\mathcal{O}\left(\log D\right)$ products $\mathbf{S}_0\left(\mathbf{S}_1\left(\cdots\left(\mathbf{S}_{Q-1}\mathbf{x}\right)\right)\right)$ between a sparse matrix and a vector, the cost of each product being $\mathcal{O}\left(D\right)$, amounting to a $\mathcal{O}(D \log D)$ time complexity.

Approximating any matrix by learning a fast transform. When the linear operator $\mathbf{W}$ is an arbitrary matrix, one may approximate it with such a sparse-product structure by learning the factors $\left\{\mathbf{S}_q\right\}_{q \in [\![Q]\!]}$ in order to benefit from a fast algorithm. [5] proposed algorithmic strategies to learn such a factorization. Based on the proximal alternating linearized minimization (`PALM`) algorithm [2], the `PALM` for Multi-layer Sparse Approximation (`palm4MSA`) algorithm aims at approximating a matrix $\mathbf{W} \in \mathbb{R}^{D \times d}$ as a product of sparse matrices by solving

$$\min_{\left\{\mathbf{S}_q\right\}_{q \in [\![Q]\!]}} \left\|\mathbf{U} - \prod_{q \in [\![Q]\!]} \mathbf{S}_q\right\|_F^2 + \sum_{q \in [\![Q]\!]} \delta_{\mathcal{E}_q}(\mathbf{S}_q), \tag{5}$$

where for each $q \in [\![Q]\!]$, $\delta_{\mathcal{E}_q}(\mathbf{S}_q) = 0$ if $\mathbf{S}_q \in \mathcal{E}_q$ and $\delta_{\mathcal{E}_q}(\mathbf{S}_q) = +\infty$ otherwise. $\mathcal{E}_q$ is a constraint set that typically imposes a sparsity structure on its elements, as well as a scaling constraint. Although this problem is non-convex and the computation of a global optimum cannot be ascertained, the `palm4MSA` algorithm is able to find local minima with convergence guarantees.

Fin du copy-pasta

In [5], the authors further propose an extension of `palm4MSA` called `Hierarchical-palm4MSA` that rely on some hierarchical optimization strategy to get better approximation results.

5 Contribution

In this paper we propose to use the `Hierarchical-palm4MSA` algorithm on the various weights matrix of a pretrained neural network so that they all are expressed as product of sparse matrices instead of dense matrices. This simple idea would allow to drastically reduce both the space complexity of any layer in the network *and* the time complexity for their computation on CPU. Indeed, even though GPU usage would definitely benefits from the space complexity saving, their highly parallelized computation wouldn't take advantage of the sparse structure of the matrices. In contrary to [8], these benefits wouldn't come at the cost of introducing bias on the choice of a fixed fast-transform structure for the weight matrices.

In convolutional layers (Equation (4)**)** The dense matrices of weights have $\mathcal{O}\left(ChwK\right)$ parameters. Expressing them as product of $Q = \log K$ sparse factors with each $\mathcal{O}\left(Chw\right)$ non-zero values allows to reduce this space complexity to $\mathcal{O}\left(Chw \log K\right)$, assuming $Chw > K$. The time complexity saving is tightly associated with this reduction of parameters: it lowers from time $\mathcal{O}\left(HWChwK\right)$ to time $\mathcal{O}\left(HWChw \log K\right)$, that is applying the fast-transform to each of the HW patches.

In fully-connected layers (Equation 2) The dense matrices of weights have $\mathcal{O}\left(Dd\right)$ parameters. Let $A := \max(D, d)$ and $B := \min(D, d)$, then expressing these dense matrices as product of $Q = \log K$ sparse factors with each $\mathcal{O}\left(A\right)$ non-zero values allows to reduce this space complexity to $\mathcal{O}\left(A \log B\right)$. The time complexity saving is once again associated to this reduction of parameters: it also lowers from $\mathcal{O}\left(Dd\right)$ to $\mathcal{O}\left(A \log B\right)$.

6 Experiments

6.1 Experimental setting

Implementation details Hardware (CPU frequency etc.) and Software (library, etc.) details

Datasets MNIST, SVHN, CIFAR10/CIFAR100, Imagenet

Models Download pretrained models from the Internet or re-train them

`palm4MSA` **algorithm settings** Sparsity level, number of factor, Hierarchical, error delta threshold, number of iteration

6.2 Number of parameters

In the whole compressed model. Compare to other methods and show that accuracy results are not compromised by the reduction of the nbr of parameter

6.3 Inference time

Show results on CPU. Maybe make a proof of concept on raspberry

6.4 Carbon footprint

It would be awesome to present results of accuracy/time versus GPU and give insight on the difference in carbon footprint between CPU and GPU. Need to read about that though. Suivre la meme technique de calcul que [7]

7 Conclusions

We worked hard, and achieved very little.

References

[1] I. D. Amodei and D. Hernandez. AI and compute. `https://openai.com/blog/ai-and-compute/`, 2018. [Online; accessed September 20, 2019].

[2] J. Bolte, S. Sabach, and M. Teboulle. Proximal alternating linearized minimization or nonconvex and nonsmooth problems. *Mathematical Programming*, 146(1-2):459–494, 2014.

[3] Y. Cheng, D. Wang, P. Zhou, and T. Zhang. A survey of model compression and acceleration for deep neural networks. *CoRR*, abs/1710.09282, 2017.

[4] A. Jassy. AWS re:Invent 2018 - Keynote. `https://www.youtube.com/watch?v=ZOIkOnW640A`, 2018. [Online; accessed September 20, 2019].

[5] L. Le Magoarou and R. Gribonval. Flexible multilayer sparse approximations of matrices and applications. *IEEE Journal of Selected Topics in Signal Processing*, 10(4):688–700, 2016.

[6] J. Morgenstern. The Linear Complexity of Computation. *Journal of the ACM*, 22(2):184–194, Apr. 1975.

[7] E. Strubell, A. Ganesh, and A. McCallum. Energy and policy considerations for deep learning in NLP. *CoRR*, abs/1906.02243, 2019.

[8] Z. Yang, M. Moczulski, M. Denil, N. de Freitas, A. Smola, L. Song, and Z. Wang. Deep fried convnets. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1476–1483, 2015.