CKN from Scratch

Kernel Methods Data Challenge MVA 23/24

Oussama Zekri
ENS Paris-Saclay
oussama.zekri@ens-paris-saclay.fr

Elyas Benyamina
ENS Paris-Saclay
elyas.benyamina@ens-paris-saclay.fr

Abstract

*As part of the image classification challenge of the course, the **CKN from Scratch** project is born ! Faced with the constraints of implementing everything from scratch and focusing on the use of kernel methods, our team set itself the challenge of implementing **from scratch**, the **Convolutional Kernel Network** introduced in [Mairal et al., 2014] and developed in [Mairal, 2016]. The results and lessons learned from this work are discussed, highlighting the potential of kernel methods in an image classification context (but not only). Our code is available on Github.*

1. Dataset and first ideas

1.1. The dataset

After a quick exploration of the dataset, we realized that it was a subset of the famous CIFAR-10 dataset. We have 10% of the (labeled) train set and 20% of the test set, i.e. $5,000$ and $2,000$ images respectively, already pre-processed.

1.2. First ideas and their limits...

Our first idea was to implement a one-vs-all multi-class SVM, with HOG features on the input datas, to boost the SVM's ability to classify. We realized that this method worked properly, without being too costly : we were getting scores of around ~ 0.40 on the leaderboard, and were even leading the challenge at the beginning.

But, as the challenge progressed, we realized that people were getting ahead of us: we felt that we could push this method, but we had the feeling that it was capped and that we couldn't guarantee that we'd obtain the best results with this framework. Then, with a little investigation in the litterature (we later noticed that there was some content on this subject at the end of the course slides), we came across two *incredible* papers...

2. Convolutional Kernel Network

2.1. Overview

The CKN framework is introduced and developped in [Mairal et al., 2014, Mairal, 2016]. Its aim is to understand and generalize the well-known CNN principle with the use of kernel machines (see Appendix B). The idea is to use kernel trick to map data characteristics onto an abstract RKHS $\mathcal{H}$. We take up the idea of convolution and pooling (for its ability to generalize well) operations, but we don't apply a non-linear activation function (e.g. ReLU), as we'll be *learning* the non-linear relationships in $\mathcal{H}$.

2.2. CKN in depth

For an in-depth understanding (with mathematical insights) of CKN, please read the Appendix A, or more simply, [Mairal, 2016].

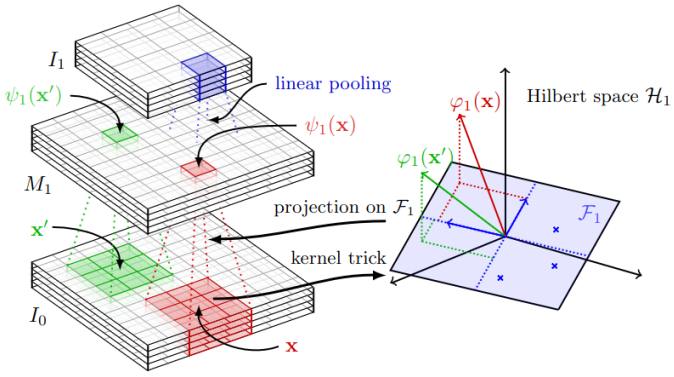

Figure 1. A Convolutional Kernel Layer. Figure from [Mairal, 2016]

3. CKN from Scratch

3.1. Implementing CKN in our context

Although in [Mairal et al., 2014, Mairal, 2016], the experiments are carried out on the CIFAR-10 dataset with very competitive scores for that time, we need to take some precautions. Firstly, we only have a subset of the dataset, and secondly, we're implementing everything **from scratch**. Because of these two constraints, we lack both data and computing power, and reproducing the same scores as in the papers seems compromised.

We use `scipy.spatial.correlate2D` (not convolve, as it saves us a matrix reflection) for convolutions, and `scipy.optimize.fmin_l_bfgs_b` for part of the optimization. Otherwise, everything is coded in Python using NumPy exclusively.

3.2. About computation time

The operation that gave us the most trouble in terms of computation time was the *correlation*, as it is very much part of the framework : we use correlations for convolutional layers, but we've also implemented linear pooling as a correlation (see Appendix A.3).

We tried to accelerate it with libraries like Numba [Lam et al., 2024], or to use the LAX-backend implementation available in JAX [Bradbury et al., 2018], but surprisingly, the algorithms

didn't run much faster than the original version (run this code, for example). We also tried to speed it up by implementing it in C++ and integrating it into our code with Cython [Behnel et al., 2011], or to implement it in CUDA to run it on the GPU, but these efforts were unsuccessful. Code snippets of these attempts are available on our Github.

4. Results

4.1. Architeture

We have implemented and tested various multi-layer CKNs. At first, we started with 5 layers. In practice, with 5 layers, we were able to train on less than 10 epochs and we didn't obtain mindblowing results, and couldn't continue with this approach. We think that we didn't have enough data to train this number of layers, or that we needed way more epochs which was seemed impossible if we wanted to keep training time below 24 hours.
We realized, thanks to [Bietti, 2022], that we could save computing time. In fact, Bietti's take is that for CIFAR-10, CKNs with 2 or 3 layers are sufficient to learn patterns properly. We therefore needed an architecture with few layers, capable of detecting patterns with a *reasonable* number of filters, since all our computations are performed on CPUs.

Still inspired by [Bietti, 2022], the architecture of our final submission is a 3-layer CKN. Each layer has filters of size 3 (which seems to be a good size for the dataset's image characteristics), a subsampling of 2, and an exponential kernel (see Appendix A.1) with parameter $\sigma = 0.6$. The first two layers have 64 filters, and the last one has 128.

The CKN is followed by a linear "fully connected" layer to classify the data. The final architecture contains $132,810$ parameters.

4.2. Optimization

Since we're not allowed to use ML librairies, we had to implement backpropagation from scratch. We also had to create a `Parameters` class to track the gradients of our parameters, which are NumPy arrays.

Since we've decided to train the CKN layers in an unsupervised way, all we need to do is perform a backpropagation on the single classification layer that follows the CKN. We used a cross-entropy loss with softmax activation, and implemented a Stochastic Gradient Descent with momentum. Optimization of the classification layer parameters is alternated between this SGD and the L-BFGS-B algorithm from `scipy.optimize.fmin_l_bfgs_b`, at each epoch.

After some experimentation, we decided to regularized our weights through a L_2 penalty, with factor $\alpha = 10^{-3}$. We also chose an initial learning of $4 \cdot 10^{-3}$ for the SGD, which we scheduled to $4 \cdot 10^{-4}$ from the 15-th epoch onwards. We ran the algorithm for 30 epochs, with a batch-size of 100 samples, and it took about ~ 24 hours.

4.3. Need for generalization

On our first attempts, obtained on a 2-layer CKN, we resulted in a score of 0.62, even though we had almost a zero error on the train set.

We immediately realized that we needed to find a way to help the model generalize better, but without data augmentation (this would inevitably have increased the amount of computations, which was already too much for our poor CPUs...). We tried a lot of approaches : L_2 regularization factor tuning, Batch Normalization, efficient Dropout for Convolutional Layers (Spatial Dropout [Tompson et al., 2015] and DropBlock [Ghiasi et al., 2018])...

Some of these approaches allowed us to gain a few percent on the results, but we didn't see any significant performance gains in our local tests. Let's note that by using a local validation set of size 200 (extracted from the original training set), we obtained results very close to the one from Kaggle, so we wanted to keep our submissions for the best attempts only.

What has enabled us to achieve significant gains was to switch from 2-layer CKN to 3-layer CKN, and to put more filters. While testing it locally, we were facing two problems, we probably need more epochs to obtain good results, and we also needed more time to train this more complex neural network. We believe that with this approach, we could have achieved even better results.

4.4. Final submission results

We were aware that this training would take some time, so as soon as we were pretty sure of our hyper-parameters (based partly on 2-layers hyper-parameter tuning and behavior of the training on first epochs on the 3-layers implementation), we launched training with 30 epochs with the full $5,000$ images training set.

Let us note that the number of epochs seemed to be an important parameter, especially for larger networks, and that we think we could have achieved better scores if we were using more epochs (but we did not have time to train for more than 30 epochs on the 3-layer CKN). We ended up scoring 0.685[1] on the public leaderboard, which corresponds to the first place.

Final scores, on public leaderboard			
$n_{lay}, n_{filt}, f_{sub}$	Kernels	Public	Private
2, (32,32), (2,6)	$\exp, \sigma = \frac{1}{\sqrt{3}}$	0.620	?
3, (64,64,128), (2,2,2)	$\exp, \sigma = 0.6$	**0.685**	?

Table 1. Final results, from Kaggle. $n_{lay}, n_{filt}, f_{sub}$ are respectively the number of layers, the number of filter for each layer and the subsampling factor for each layer

5. Conclusion

Obtaining these results for this challenge was very difficult. We had to implement the CKN from scratch, optimizing the various parameters with limited attempts.

We'd like to mention Bietti's PhD thesis [Bietti, 2019], and the work he (and others !) continue to do in this direction, in particular in [Bietti, 2022]. We'd also like to mention Mairal's first CuDNN/Matlab implementation, Chen's Pytorch implementation and Bietti's C++ implementation, which helped us a lot understanding the concepts and details behind the CKN framework.

Although our CKN implementation is by no means optimal, since it was implemented without using any ML libraries, it took a lot of time and dedication to get everything running and producing acceptable results.

The challenge of coding everything from scratch has taught us a lot about the notions of the Kernel Methods course, but also and above all about Machine Learning concepts, and ideas that are only fully understood when implemented from scratch.

We're also delighted to have been able to gain an in-depth understanding of Mairal's invention, the Convolutional Kernel Network.

References

[Behnel et al., 2011] Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., and Smith, K. (2011). Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39. 2

[Bietti, 2019] Bietti, A. (2019). *Foundations of deep convolutional models through kernel methods*. Theses, Université Grenoble Alpes. 2

[Bietti, 2022] Bietti, A. (2022). Approximation and learning with deep convolutional models: a kernel perspective. 2, 4

[Bietti and Mairal, 2018] Bietti, A. and Mairal, J. (2018). Group invariance, stability to deformations, and complexity of deep convolutional representations. 5

[Bradbury et al., 2018] Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. (2018). JAX: composable transformations of Python+NumPy programs. 1

[Ghiasi et al., 2018] Ghiasi, G., Lin, T.-Y., and Le, Q. V. (2018). Dropblock: A regularization method for convolutional networks. 2

[Lam et al., 2024] Lam, S. K., stuartarchibald, Pitrou, A., Florisson, M., Markall, G., Seibert, S., Self-Construct, E., Anderson, T. A., Leobas, G., rjenc29, Collison, M., luk-f a, Bourque, J., Kaustubh, Meurer, A., Oliphant, T. E., Riasanovsky, N., Wang, M., densmirn, KrisMinchev, Masella, A., Pronovost, E., njwhite, Wieser, E., Totoni, E., Seefeld, S., Grecco, H., Peterson, P., Virshup, I., and G, M. (2024). numba/numba: Numba 0.59.1. 1

[Mairal, 2016] Mairal, J. (2016). End-to-end kernel learning with supervised convolutional kernel networks. 1, 4, 5

[Mairal et al., 2014] Mairal, J., Koniusz, P., Harchaoui, Z., and Schmid, C. (2014). Convolutional kernel networks. 1

[Paulin et al., 2016] Paulin, M., Mairal, J., Douze, M., Harchaoui, Z., Perronnin, F., and Schmid, C. (2016). Convolutional patch representations for image retrieval: an unsupervised approach. 5

[Tompson et al., 2015] Tompson, J., Goroshin, R., Jain, A., LeCun, Y., and Bregler, C. (2015). Efficient object localization using convolutional networks. 2

A. Convolutional Kernel Network in depth

A.1. Dot product kernels

Let us consider a set $\mathcal{X}$, a p.d.kernel $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, its associated RKHS $\mathcal{H}$ and its associated mapping fonction $\phi : \mathcal{X} \to \mathbb{R}$, i.e. the function such that $\forall x, x',\ K(x, x') = \langle \phi(x), \phi(x') \rangle_{\mathcal{H}}$.

Let us define homogeneous dot product kernels, that will be useful in the following.

Definition A.1 ((Homogeneous) Dot product kernels) *Let us consider some function $\kappa : \mathbb{R} \to \mathbb{R}$. A dot product kernel is a p.d. kernel $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ defined as*

$$\forall x, x' \in \mathcal{X}, K(x, x') = \kappa(\langle x, x' \rangle)$$

We also define homogeneous dot product kernels as

$$\forall x, x' \in \mathcal{X}, K(x, x') = \|x\| \|x'\| \kappa \left(\left\langle \frac{x}{\|x\|}, \frac{x'}{\|x'\|} \right\rangle \right)$$

If κ has some regular properties (see [Mairal, 2016]), we can use homogeneous dot product kernels to represent local image neighborhoods in $\mathcal{H}$.

A lot of these kernels exist, see [Bietti, 2022] for details. In our work, we use in particular the *exponential* dot product kernel $K_{\exp}(x, y) = \mathrm{e}^{\beta(\langle x, x' \rangle - 1)} = \mathrm{e}^{-\frac{\beta}{2}\|x - x'\|_2^2}$, where β is a parameter associated. With $\beta = \frac{1}{\sigma^2}$, we recover the well-known Gaussian Kernel. In our experiments in Section 4, we use this σ parameter.

A.2. Mathematical insights and computability

Now, we can define mathematically an image as a mapping $I_0 : \Omega_0 \to \mathbb{R}^3$ (as an image has 3 color channels). If x and x' are two patches extracted from I_0 (and $\phi_1(x)$ and $\phi_1(x')$, their representation in $\mathcal{H}$), we can use Nyström method to approximate $\phi_1(x)$ and $\phi_1(x')$ by their projection $\psi_1(x)$ and $\psi_1(x')$ onto a **finite** dimensional subspace $\mathcal{F}$ (see Figure 2).

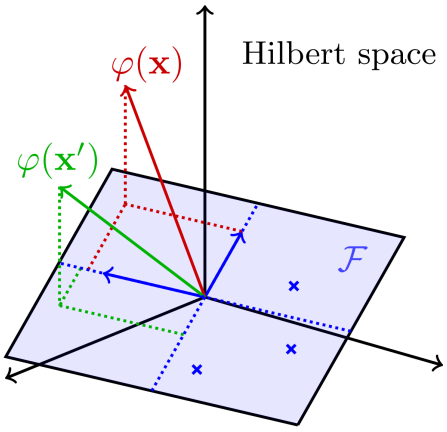

Figure 2. Illustration of Nyström method to "reduce the dimension". Figure from slide 702 of the course slides

The subspace $\mathcal{F}$ is defined as $\mathcal{F} = \mathrm{span}(z_1, \ldots, z_p)$, where $(z_i)_{i \in \{1 \ldots p\}}$ are anchor points, which of norm 1. The subspace can be optimized in both a supervised and an unsupervised way.

What is powerful is that we are moving from a potential infinite dimensional space, to a finite dimensional subspace, which is interesting for numerical computations. Also, [Mairal, 2016] mentions that we obtain really good results for the unsupervised method, with Spherical KMeans. That's why we decided only implement this unsupervised way in our code.

A.3. The Convolutional Kernel Network

We can therefore construct our Convolutional Kernel Layer (see Figure 3). The idea is that we first extract patches from the image I_0 and normalize them. Then, we convolve them and use the kernel trick to represent these patches in $\mathcal{H}$ with ϕ, and we project them on $\mathcal{F}$ as ψ. We can therefore compute the pooling operations. After the that operation, we have constructed a "feature map" $I_1 : \Omega_1 \to \mathbb{R}^{p_1}$, that can be re-used.

Note that, (gaussian) linear pooling is defined as

$$I_1(x) = \int_{x' \in \Omega_0} M_1(x') \mathrm{e}^{-\beta \|x - x'\|_2^2} \mathrm{d}x'$$

where M_1 is the "feature map" after the convolution operation. That is why, we can interpret the pooling operation as a "convolution" operation.

We can now build a **Convolutional Kernel Network** by stacking Convolutional Kernel Layers, and we will have a *representation* of the image at the output.

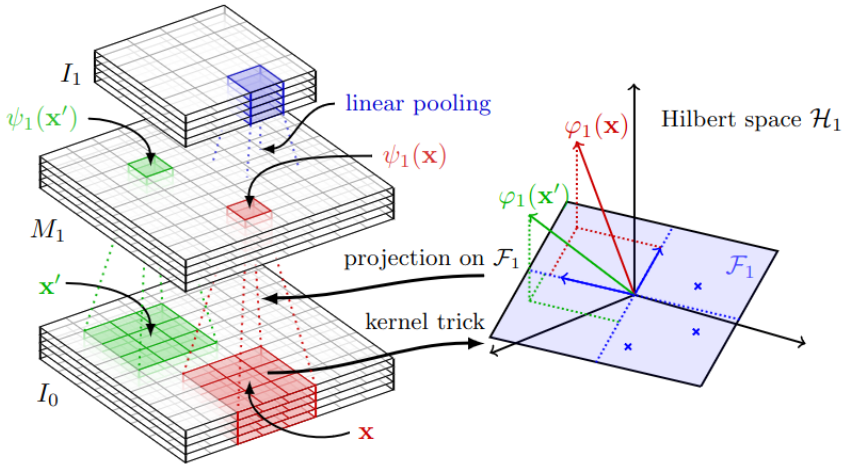

Figure 3. A Convolutional Kernel Layer. Figure from [Mairal, 2016]

B. Convolutional Neural Network vs Convolutional Kernel Network

In [Bietti and Mairal, 2018], it is shown that CKNs contain a large class of CNNs with smooth homogeneous activation functions.

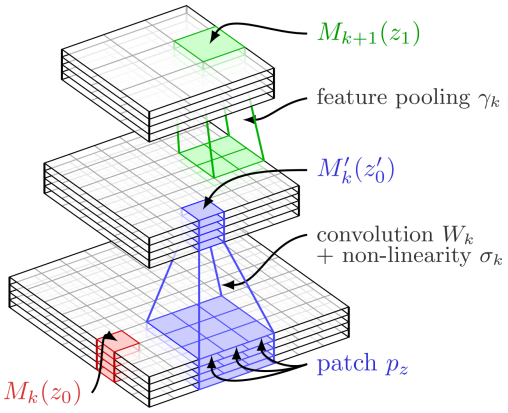

Figure 4. A classical CNN. Figure from [Paulin et al., 2016]

The similarities and differences between CKN and CNN are well illustrated in Figures 3 and 4.

On the one hand, A CNN of K layer can be represented by its output $f_{CNN}(x)$, if x is the input, defined as :

$$f_{CNN}(x) = \gamma_K(\sigma_K(W_K \dots \gamma_2(\sigma_2(W_2\gamma_1(\sigma_1(W_1x)) \dots))$$

where $(W_k)_k$ represent the convolution operations, $(\sigma_k)_k$ are pointwise non-linear functions, (e.g., ReLU), and $(\gamma_k)_k$ represent the pooling operations (see [Paulin et al., 2016]).

On the other hand, A CKN of K layer can be represented by its output $f_{CKN}(x)$, if x is the input, defined as :

$$f_{CKN}(x) = \gamma_K(\sigma_K(W_K(P_K \dots \gamma_2(\sigma_2(W_2(P_2(\gamma_1(\sigma_1(W_1(P_1(x)) \dots))$$

where $(P_k)_k$ represent the patch extraction and normalization, $(W_k)_k$ represent the convolution operations, $(\sigma_k)_k$ are kernel operations (it allows us to learn non-linearity in the RKHS), and $(\gamma_k)_k$ represent the pooling operations.

An excellent reference for understanding everything about this is [Bietti and Mairal, 2018].