Alma Mater Studiorum · Università di Bologna

Scuola di Scienze
Dipartimento di Fisica e Astronomia
Corso di Laurea Magistrale in Fisica

Scattering Networks: Efficient 2D Implementation And Application To Melanoma Classification

Relatore:
Prof. Renato Campanini

Correlatore:
Dott. Matteo Roffilli

Presentata da:
Eugenio Nurrito

Anno Accademico 2015/2016

Abstract

Machine learning is an approach to solving complex tasks. Its adoption is growing steadily and the several research works active on the field are publishing new interesting results regularly.

In this work, the scattering network representation is used to transform raw images in a set of features convenient to be used in an image classification task, a fundamental machine learning application. This representation is invariant to translations and stable to small deformations. Moreover, it does not need any sort of training, since its parameters are fixed and only some hyper-parameters must be defined.

A novel, efficient code implementation is proposed in this thesis. It leverages on the power of GPUs parallel architecture in order to achieve performance up to 20× faster than earlier codes, enabling near real-time applications. The source code of the implementation is also released open-source.

The scattering network is then applied on a complex dataset of textures to test the behaviour in a general classification task. Given the conceptual complexity of the database, this unspecialized model scores a mere 32.9 % of accuracy.

Finally, the scattering network is applied to a classification task of the medical field. A dataset of images of skin lesions is used in order to train a model able to classify malignant melanoma against benign lesions. Malignant melanoma is one of the most dangerous skin tumor, but if discovered in early stage there are generous probabilities to recover. The trained model has been tested and an interesting accuracy of 70.5 % (sensitivity 72.2 %, specificity 70.0 %) has been reached. While not being values high enough to permit the use of the model in a real application, this result demonstrates the great capabilities of the scattering network representation.

Sommario

Il *machine learning* è un approccio alla risoluzione di problemi complessi. Il suo utilizzo è in costante crescita e i diversi lavori di ricerca attivi nel settore pubblicano regolarmente nuovi, interessanti risultati.

In questo lavoro si adopera la rappresentazione *scattering network* per trasformare immagini grezze in un insieme di *features* utilizzabili per la classificazione di immagini, un'applicazione fondamentale del *machine learning*. Questa rappresentazione è invariante per traslazioni ed è stabile alle piccole deformazioni. Non necessita inoltre di alcun tipo di addestramento, poiché i suoi parametri sono fissi e solo alcuni iper-parametri devono essere definiti.

In questa tesi è proposta una nuova ed efficiente implementazione del codice. Essa sfrutta la potenza dell'architettura parallela delle GPU per raggiungere performance fino a 20 volte più veloci dei codici precedenti, consentendo la realizzazione di applicazioni quasi in tempo reale. Il codice sorgente dell'implementazione è inoltre rilasciato *open-source*.

La *scattering network* è poi applicata ad un dataset complesso di texture per testarne il comportamento in un problema generico di classificazione. Data la complessità concettuale del database, questo modello non specializzato raggiunge un mero 32.9 % di accuratezza.

Infine, la *scattering network* è applicata ad un problema di classificazione in ambito medico. Un dataset di immagini di lesioni della pelle è utilizzato per addestrare un modello che possa classificare melanomi maligni contro lesioni benigne. Il melanoma maligno è uno dei più pericolosi tumori della pelle, ma, se scoperto in uno stadio precoce, le probabilità di cura sono elevate. Il modello addestrato è stato testato ed è stata raggiunta un'interessante accuratezza del 70.5 % (sensibilità 72.2 %, specificità 70.0 %). Pur non essendo valori abbastanza elevati da permettere l'utilizzo del modello in un'applicazione reale, i risultati dimostrano le grandi possibilità della rappresentazione *scattering network*.

Contents

CHAPTER 1

Introduction

In the last decade, the world faced with the concept of "big data". This term refers to the great mole of organized, accessible and growing information coming from assorted groups of measures. The introduction of this concept was made possible by the incessant evolution of electronics and information technology. Electronics contributed making available cheap and fast sensors for every needing: imaging, chemical analysis, physical measures are only some examples. Information technology, in its turn, provided the software and hardware platforms to analyze obtained information. New devices, with different architectures (CPU, GPU, DSP, FPGA, ...) and with increasing power in terms of FLOPS, are available every day.

Many approaches were used during the years to analyze this data. The ultimate goal is to use data to understand a trend and, consequently, make predictions and take decisions. Statistics has always been used for this purpose. But even if simple statistics approaches were usually enough to deal with a limited amount of controlled data in the past, the challenge of analyzing "big data" is to extrapolate information in situations that were not represented by available data, thus to be able to *generalize*.

"Machine learning" is a term that can be heard around every day. This approach consists in training a model to make predictions basing on parameters learned from available data. Classical approaches, intead, have the difference of having little or none information of data included in the structure of the model. Therefore, machine learning algorithms are able to generalize, and well.

Many applications are now based on machine learning and the medical field is no exception. Histological and cytological images can be analyzed by an algorithm that detects and classifies each single cell. This helps physicians to classify microscopy samples faster. CAD systems (Computer Aided Detection/Diagnosis system) based on machine learning are present in many areas, for example to automatically recognize masses and microcalcifications in mammograms [1].

In this thesis, the “scattering convolutional network” concept is used. This method, proposed by S.Mallat and J.Bruna in [2, 3], allows to create a representation of a signal (in our case an image) that is invariant to translations and stable to deformations. Backed by the important results obtained by the *wavelets* proposed by the same Mallat, the scattering convolutional network offers an interesting way to extract features of a signal, that should allow a machine learning algorithm to learn a generalization of the problem. Chapter 2 starts with an introduction to what a “machine learning algorithm” is and then explains the scattering convolutional network algorithm.

One of the contributions of the thesis work is the realization of the scattering network algorithm in code capable of running on GPUs, thus improving time performance over the original serial code. Chapter 3 shows how an efficient implementation has been studied over several revisions, with analysis of compatibility tests and the final performance improvement. Moreover, the written source code has been released open source with the hope to help other researchers in future studies or applications with the scattering network algorithm.

The interest then moved to classification. In this task, the theory of scattering networks suggests that they are able to create a discriminable representation with important stabilities. Chapter 4 presents the general methodology used to train and test a classifier, with punctual details on the actual procedure used in the following two applications.

Since studies on scattering networks appeared to work well in recognizing patterns and textures, the algorithm has been firstly tested in Chapter 5 to extract features from the recent “Describable Textures Dataset” [4]. This application was mainly a first attempt to understand the possibilities of the algorithm toward a more practical application.

A medical field where machine learning is still at an early stage is dermatology. A particular task where it is gradually acquiring interest is melanoma classification. The investment on this problem arises from the possibility to cure by biopsy a malignant melanoma if it is detected in the early phase of development. Therefore, if an automatic system could perform a preliminary diagnosis, possible malignant melanoma could be discovered in time to be treated. Chapter 6 sets out an attempt to classify a skin lesion (nevi) to be either a benign lesion or a malignant melanoma. This task is dealt with a model based on a scattering network representation and a SVM classifier. A recent valuable database from the ISDIS (International Society for Digital Imaging of Skin) [5] is used. The main intention of this application was not to try to reach a state-of-the-art comparable results, but to test if the general purpose scattering network algorithm could give interesting results in an application of practical utility and of common interest.

The work of thesis was mainly developed in Bioretics’ offices[6], a start-up company whose primary interest is solving imaging and computer vision problems.

CHAPTER 2

Machine Learning and Scattering Convolutional Networks

Machine learning is the field of computer science that tries to create algorithms that are able to learn from a set of data. Mitchell [7] wrote a definition of a learning algorithm:

Definition 2.1. *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks T, as measured by P, improves with experience E.*

We can recognize in the experience E the big data available nowadays, that can spread over a great range of fields. The concept of performance measure P is also very broad, as it ranges from objective metrics to individual judgments. Nevertheless, measure P is usually specific to the task T. Tasks, instead, can be usually grouped by their type. Most common kinds of task are classification, regression, clustering, density estimation and feature reduction.

In this work, we will deal mainly with classification. A classification task consists of assigning an input sample to a category (also called class, whose name is defined label). Thus, a machine learning algorithm needs to learn a function $f : \mathbb{R}^n \to \{1, \dots, k\}$, which map the n dimensional input to one of the k categories. A common, but complex, example of classification is image recognition, in which the algorithm is required to output a label for the input image.

At the baseline, a classification task consists of a measure of similarity. Simple Euclidean distance is generally unable to discriminate and give good measures of similarities on real world data where intra-class variability is relevant.

Continuing with the example of image classification, there is a great variability in images belonging to the same class, due to rigid and non-rigid transformations. In fact, a simple operation like a translation can create a great variation of the value of each pixel of the image. This intra-class variability is an obstacle to classification and must be eliminated in order to have a more informative measure of similarity. The general approach to this problem is based on Kernel Methods [8], who consist of projecting input data x in a new representation $\Phi(x)$. This could allows to build a representation that is invariant to some transformations. History of physics teaches that most modern theory are founded on the presence of symmetries and thus invariants in a system. Having a representation of this could easily lead to good subsequent results.

In the image field, rigid transformations such as translations, scaling and rotations are generally uninformative for classification. Therefore a good representation should be invariant to this kind of deformations. In fact, they are usually caused only by the relative pose of the object with respect to the camera, an information that in most application is in inconspicuous. On the other side, non-rigid deformations may induce both intra-class and inter-class variability. For example, a non-rigid deformation could transform a “1” into a “7”, and thus changing the outcome of a digits recognition system.

A good representation $\Phi(x)$ for classification should therefore not be deformation invariant, but stable to small deformations [3]. This will allow to discriminate between different classes with a kernel classifier. In fact, the kernel will permit to handle the small deformations.

The stability to deformations is expressed as a Lipschitz continuity condition. Given a deformation $L_\tau x(u) = x(u-\tau(u))$ where x is an image, u the position inside the image and $\tau(u)$ the function that gives the deformation, we can express the gradient of the transformation $\nabla\tau(u)$, whose norm $|\nabla\tau(u)|$ measures the amplitude of the deformation. Therefore, a representation $\Phi(x)$ is stable to deformations if it is Lipschitz continuous relative to the deformation metric:

$$\|\Phi(L_\tau x) - \Phi(x)\| \leq C\|x\| \sup_u |\nabla\tau(u)|$$

where $\|x\| = \int |x|^2 du$.

Getting rid of variance due to small deformations, a good representation must preserve (and enhance) the discriminability that lays in information given by high-order moments.

2.1 Toward Scattering Convolutional Networks

The problem is to find a “good” representation $\Phi(x)$ for the data, as expressed before. It should be stable to additive noise and small deformations and it should carry information of high-order moments for discriminability. In the treatment we will also impose the stronger requirement to the representation to be (locally) translation invariant. Features for this representation could be extracted by filtering with wavelet filters the input signal.

Wavelets are wave-like functions localized in both space and frequency [9] and thus they are stable to deformations. Given a single band-pass wavelet $\psi(u)$, it is possible to build a bank of multiscale directional wavelets:

$$\psi_{2^j r} = 2^{2j}\psi(2^j r^{-1}u)$$

where $j \in \mathbb{Z}$ and $r \in G$, with G a finite rotation group in $\mathbb{R}^2$. Let's denote $\lambda = 2^j r$ for a simpler notation. An example of a filters bank of this kind can be seen in Figure 2.1.

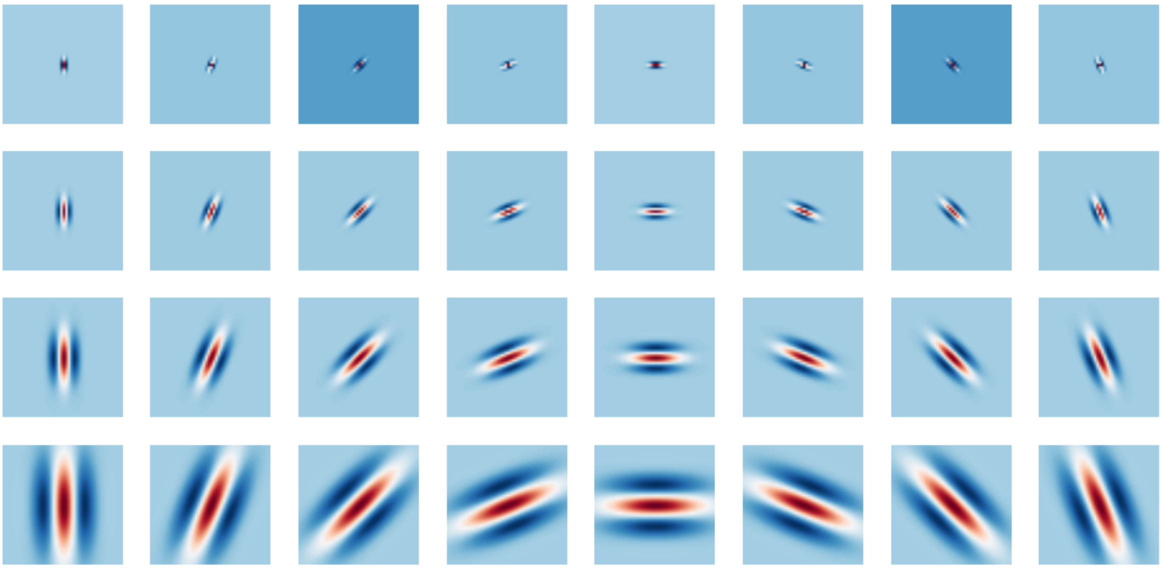


Figure 2.1: A filter bank of complex Gabor wavelets with 4 scales and 8 orientations. Only the real part is displayed.

Given a family of wavelets $\{\psi_\lambda(u)\}_\lambda$, the operation of convolving a signal x with each wavelet ψ_λ is called wavelet transform and is defined $\{x \star \psi_\lambda\}_\lambda$ where each element of the set is called wavelet coefficient.

If the wavelet set covers the entire frequency plane (as illustrated in Figure 2.2), the wavelet transform is stable to deformations and invertible.

In particular, the wavelet transform is linear, so it is translation covariant, not invariant. [3] shows that a non-linearity is needed to build a translation invariant representation. This is chosen to be the modulus operation, a point-wise non-linearity that preserves deformation stability and conserves the energy. Conceptually, the modulus smooths the complex wavelet coefficients, pushing high frequencies to lower ones.

$$U[\lambda]x = |x \star \psi_\lambda|$$

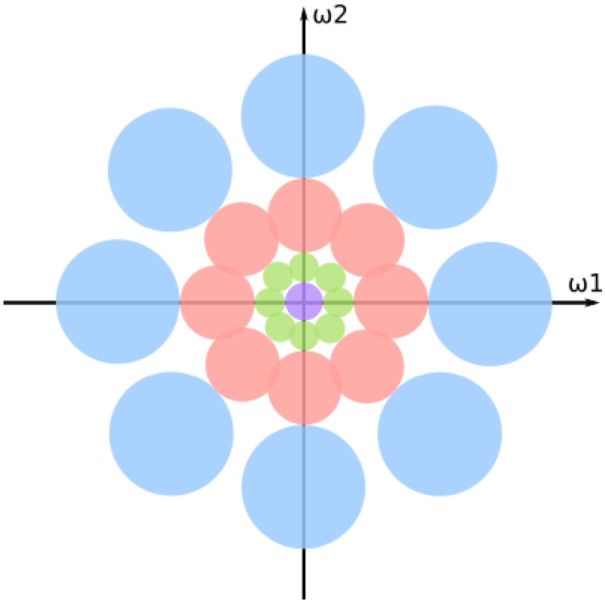


Figure 2.2: Representation of a uniform and complete covering of the frequency plane by a set of scaled and rotated wavelets.

A translation invariant representation is obtained by applying the only linear operator stable to translation: the average. For the classification task, it's often better to have a local stability. This means having a local averaging, i.e. a blurring operator ϕ_{2^J} on a spatial window of size 2^J. The result is a translation invariant wavelet coefficient:

$$S_J[\lambda]x = |x \star \psi_\lambda| \star \phi_{2^J}$$

Unluckily, this operation will kill the high-order moments that lead to discriminability. Lost information can be recovered by applying this same procedure to $U[\lambda]x$, obtaining:

$$S_J[\lambda_1, \lambda_2]x = ||x \star \psi_{\lambda_1}| \star \psi_{\lambda_2}| \star \phi_{2^J}$$

where ψ_{λ_1} and ψ_{λ_2} belong to the same wavelets family. This procedure can be applied iteratively to extract wavelet coefficients of higher order moments.

Any sequence $p = (\lambda_1, \lambda_2, ..., \lambda_m)$ is called a "path" of length m, hence:

$$S_J[p]x = ||||x \star \psi_{\lambda_1}| \star \psi_{\lambda_2}|...| \star \psi_{\lambda_m}| \star \phi_{2^J}$$

This approach naturally creates a convolutional network structure.

2.2 Scattering Convolutional Network

The scattering convolutional network is the representation composed of all coefficients $S_J[p]x$ for paths p that have a length $m \leq m_{max} = M$.

Given a signal x, all coefficients $S_J[p]x$ are obtained from a layered structure, where the scattering propagator $U_Jx = \{x \star \phi_{2^J}, |x \star \psi_\lambda|\}_\lambda$ is iteratively applied to obtain the path p. An illustration of a scattering convolutional network can be seen in Figure 2.3.

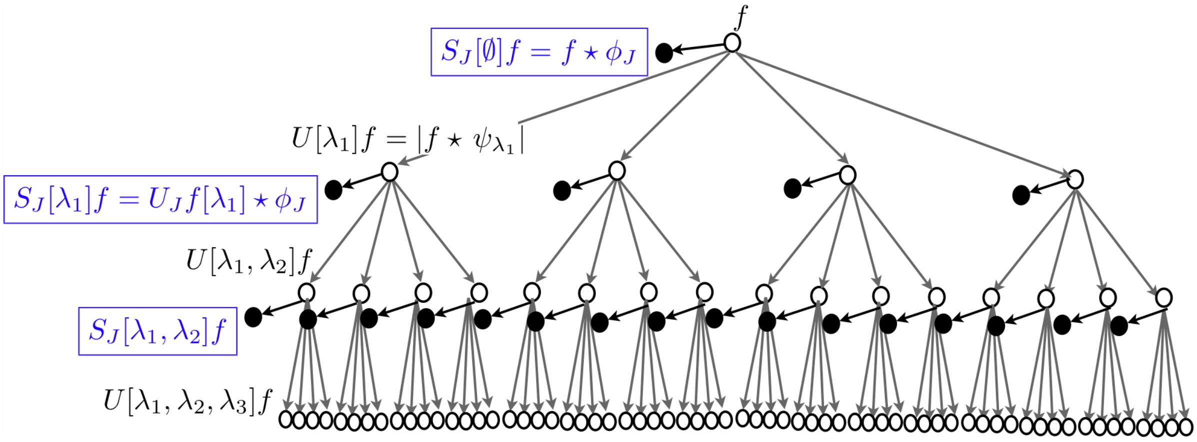

f
$S_J[\emptyset]f = f \star \phi_J$
$U[\lambda_1]f = |f \star \psi_{\lambda_1}|$
$S_J[\lambda_1]f = U_Jf[\lambda_1] \star \phi_J$
$U[\lambda_1, \lambda_2]f$
$S_J[\lambda_1, \lambda_2]f$
$U[\lambda_1, \lambda_2, \lambda_3]f$

Figure 2.3: A graph representation of a scattering neural network from [2]. Scattering propagator U_J is applied to f to compute $U[\lambda_1]f$ and outputs $S[0]f$. U_J is then applied to every $U[\lambda_1]$ to obtain the second layer of the scattering and relative outputs. This iteration is repeated up the required depth.

The signal representation offered by scattering networks has several important properties.

First of all, no parameter needs to be learned to perform the transformation. Only some hyper-parameters could be selected with some tests. This is opposed to common convolutional neural networks, where the filters need to be learned from the data with the back-propagation algorithm.

Secondly, it is invariant to translations up to 2^J because of the final averaging windows. This property is essential in image classification where a small translation rarely influences the category.

Then, as previously stated, with a suitable set of wavelets the representation is stable to deformations. This guarantees that small deformations that shouldn't affect classification result can be completely managed by a kernel classifier. Some further works on the scattering network included stability to other particular transformations. For example, in [10] a further stability to rotations is inserted in the structure of the representation by combining signals scattered from different rotations of the filters. Other stabilities could be included if the mathematical formulation could be given. Beside that, this work will only deal with the classic formulation of the scattering network.

Finally, it has been proved that, with a proper set of wavelet, the scattering operator U_Jx is contractive, in the sense that $\|U_Jx - U_Jy\| \leq \|x - y\|$ and it also preserves signal norm $\|U_Jx\| = \|x\|$. Therefore it can be shown that energy of last layers converges to zero when m_{max} increases. This is important in numerical applications because it allows to keep the network depth low with a negligible loss of energy.

A common set of wavelets used for the scattering network representation is based on scaled and rotated Gabor Wavelets (an example is shown in Figure 2.1). These wavelets are basically a Gaussian function modulated by a complex exponential. Other kinds of wavelets have been used, for instance Haar wavelets [11].

Considering Gabor wavelets, an analogy with the visual system of mammals can be proposed. In fact, Hubel and Wiesel [12] and subsequent studies suggested that primary visual cortex V1 is formed by neurons whose response has the same kind of shape of Gabor wavelets. These neurons are organized in hypercolumns (Figure 2.4), and the position of each neuron in the column defines the orientation, the scale and the location of the patterns of light that it recognizes. Scattering convolutional networks follows a similar approach, decomposing the input signal by the response of scaled and rotated Gabor wavelets, although the translation (location) is chosen to be an invariant by construction.

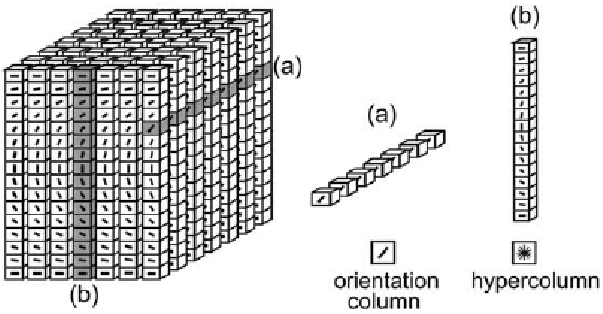


Figure 2.4: An illustration of hypercolumns in the primary visual cortex. Each box is a neuron that responds to a particular orientation and scale of light patterns. From [13].

2.3 The Algorithm

Illustration of Figure 2.3 suggests to calculate the scattering network representation iterating layer by layer, i.e. increasing the length of paths p.

Let's call Λ_J^m the set of all paths $p = (\lambda_1, ...\lambda_m)$ of length m. We define also the set of all U and S_J computed for paths of length m as $U[\Lambda_J^m] = \{U[p]x\}_{p \in \Lambda_J^m}$ and $S_J[\Lambda_J^m] = \{S_J[p]x\}_{p \in \Lambda_J^m}$. Finally let's define the set $\Lambda_J = \{\lambda = 2^j r : r \in G^+, j \in \mathbb{Z}, j \geq -J\}$

The scattering network representation of a signal x can be computed with the base Algorithm 1.

The algorithm is iterative and uses previously calculated $U[\Lambda_J^{m-1}]$ to compute new data at layer m. In this way, each path p is composed step by step to avoid the evaluation of the same expressions multiple times.

Algorithm 1 Scattering Network Algorithm

```
$U[\Lambda_J^0] \leftarrow x$
for $m = 1$ to $M$ do                                          ▷ for each layer
    Compute every $U[\Lambda_J^m] = \{|U[\Lambda_J^{m-1}] \star \psi_\lambda|\}_{\lambda \in \Lambda_J}$   ▷ band-pass filtering and modulo
    Output every $S_J[\Lambda_J^m]x = U[\Lambda_J^{m-1}]x \star \phi_{2^J}$        ▷ output low-pass filtering
end for
```

This version of the algorithm is also expressed in a very synthetic style. A real implementation of the algorithm should make sure to iterate over every path p and every filter ψ_λ. If the value of parameters M, J and L grows, the number of operations to perform increase exponentially with M. It can be seen that the total number of elements $U[p]$ and $S_J[p]$ is $1 + (J \cdot L)^M$, giving a total of $2 \cdot [1 + (J \cdot L)^M]$ convolution operations.

Starting from this pseudo-code, a real implementation of the scattering network algorithm can use several expedients to achieve good performance. Some of theme are exposed in subsequent paragraphs.

2.3.1 *Downsample*

When a signal is passed through a bandpass or a lowpass filter, it may be possible to uniformly downsample it without losing information, according to the Shannon-Nyquist's rule. For example, a half band lowpass filter would allow to discard every other sample, because after the filtering the signal has the highest frequency half of the original one (as illustrated in Figure 2.5). This allows to remove redundant information with the benefit of having a more compact representation.

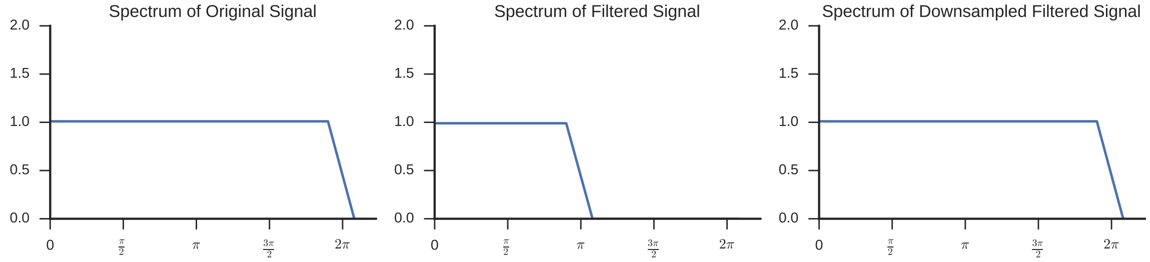


Figure 2.5: An illustration of the FFT spectrum after a low-pass filtering (center) and a downsample (right).

In the scattering network, both ψ_{2^J} and ϕ_λ are filters that lower the frequency band of the signal (except when j of λ is 0). The averaging with ψ_{2^J} allows to downsample $S_J[p]x$ at intervals 2^J. Instead, band-pass filters ϕ_λ reduce frequency band with respect to the scale j of the filter, therefore $U[p]x$ can be sampled at intervals 2^j.

Using this technique, if the signal x is an image of N pixels, the coefficient $S_J[p]x$ will have only $2^{-2J}N$ elements.

The application of this method to the algorithm previously described will allow to compute simpler operations. Indeed, the signals size will decrease through paths. However, some care must be used to decide the right sample interval at each step of each path. Final signals must not be subsampled at intervals more than 2^J in total.

2.3.2 *Pre Computing Filters*

The wavelet filters bank $\{\psi_\lambda(u)\}_{\lambda\in\Lambda_J}$ is fixed and reused several times during the execution of a scattering network. Therefore, a major optimization is to pre-calculate the filters from the required hyper-parameters M, J and L of the scattering network. As this operation could take a remarkable amount of time, wavelet filters banks could also be cache stored on disk to improve performance in multiple runs.

Filters can be generated in the spatial space or in the frequency space. The first method does not need to know the original image size to create filters. Instead, in order to create filters directly in the Fourier space, a fundamental requirement is the previous knowledge of the image size. This way, the filter generator can create the appropriate filters for each possible subsample of the original signal throughout the scattering paths of the network.

2.3.3 *Reduced Scattering Transform*

In [3], authors showed that scattering energy is concentrated along frequency-decreasing paths p. This means that all paths where index j decreases can be omitted from computation with negligible loss of energy and thus quality of the representation for classification. Instead, this approach allows to implement a noticeably faster algorithm, where lots of paths are discarded (in particular those with low j that contains larger signals in the downsample approximation, which are the ones that would require more time to be computed). The resulting representation is called “Reduced Scattering Transform” and derives directly from the original algorithm by adding checks on which λ to use to compute new $U[\Lambda_J^m]$.

2.3.4 *Convolution Filtering and Convolution Theorem*

The convolution operation is clearly the most present and time consuming operation in the proposed algorithm.

Let’s recall from [14] that given two continuous functions $f(t)$ and $h(t)$ of the continuous variable t, the convolution is defined as:

$$f(t) \star h(t) = \int_{-\infty}^{\infty} f(\tau)h(t-\tau)d\tau$$

Applying the Fourier transform to $f(t) \star h(t)$ gives the Fourier pair:

$$f(t) \star h(t) \Leftrightarrow F(\omega)H(\omega)$$

and a similar procedure would also give:

$$f(t)h(t) \Leftrightarrow F(\omega) \star H(\omega)$$

These two pairs are called the Convolution Theorem. It states that a convolution in one of the spaces is equivalent to a point-wise multiplication in the other space.

The convolution theorem applies also to discrete signals, but periodicity problems may arise. Discrete functions have a periodic Fourier spectrum, as explained by the "sampling theorem", so the convolution with the convolution theorem would give a periodic convolution, also called "circular convolution". This creates a "wrap-around error" at boundaries, where data from adjacent periods interfere. A simple way to get rid of this error is an appropriate padding, as explained in the next section.

As explained in [15], performing convolution with the convolution theorem is known as "fast convolution" because the problem is $O(N \log N)$ complex due to the Fast Fourier Transform, whereas standard convolution is (at least) $O(N^2)$. In practical implementations, there is not a commonly shared approach toward the problem because peculiarities of different architectures (CPU, GPGPU, dedicated hardware, ...) can be exploited in each approach.

The "fast convolution" approach definitely offers best performance for large signals and filters, but it has some downsides too, such as the "wraparound error" problem, a relatively lower numerical precision and a considerable larger memory requirement.

Nevertheless, this approach has been used in our implementation because some details of the scattering network algorithm can boost performance with this approach, as further explained in Chapter 3.

2.3.5 *Padding*

When talking about convolution, a major problem arises at borders. The issue is that a finite discrete signal (e.g. an image) is not defined outside its boundaries. Therefore the convolution with a filter is undefined outside the borders of the signal.

Several ways to handle this problem had been proposed and each one rely on some requirements of the problem or knowledge of the data outside boundaries [16, 17].

A first approach is to limit output region only to those pixels that are far enough from borders to have the convolution value exactly defined. In this way, output image have a smaller size than input image by the filter size minus one ("Blurring with cropping" in Figure 2.6).

The opposite method is to pad the input image, i.e. to add a border around the input image, so that pixels outside boundaries are actually defined. There are different methods to establish the value of those pixels.

The simplest one is to add zeros ("zero-padding") outside the image. This approach is useful if the signal is supposed to go to zeros outside the region where it is defined. A variation is to pad with the supposed value, different from zero. For any variation, many discontinuities are created at borders. This method can create dark borders around convolved image as seen in Figure 2.6.

If the signal is supposed to be periodic, the "wrap-padding" will add paddings that are copies of the image on the other side of the axis, like a modulo operation on the position. This method also creates discontinuities.

Finally, a common approach with real-world image is the "symmetric-padding", where the padding is a reflection with respect to the border. Discontinuities of the first derivative are created at borders but this approach usually creates nice-looking borders on images.

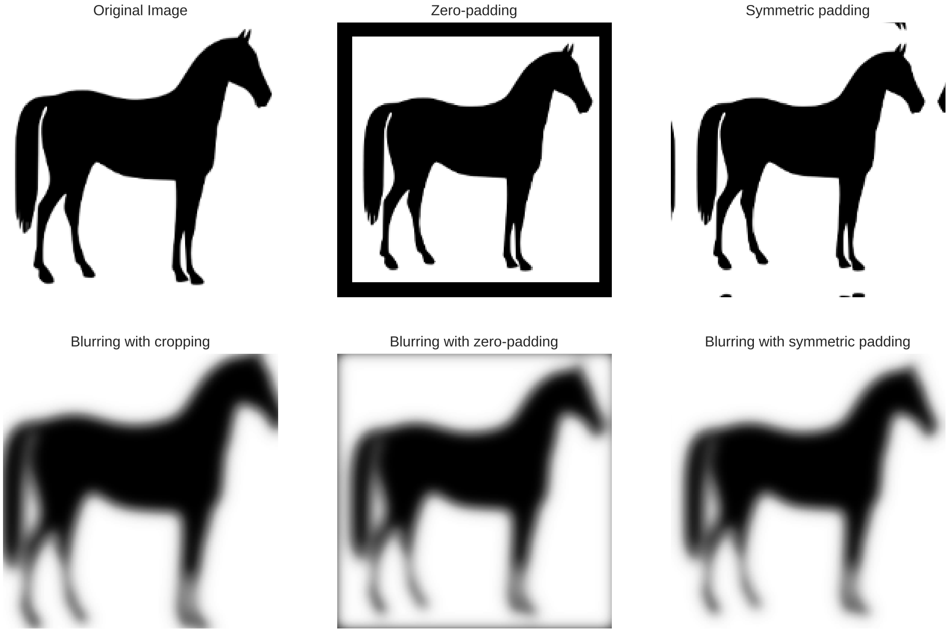


Figure 2.6: Illustration of several kind of padding and the result of a convolution with a Gaussian blurring filter.

CHAPTER 3

GPU Implementation of Scattering Convolutional Networks

One of the main purpose of this thesis is to implement the scattering network algorithm with a code that can run efficiently on a GPGPU.

The planning of the code has been guided by some objectives.

First of all, the implementation needed to be easy and fast to prototype. At the same time, high performance were required to obtain interesting results. Therefore the Python language has been chosen for its interpreted scripting nature that allows to prototype fast, the generous number of highly optimized, open source modules, and the previous experience of the author with this programming language. For the GPU side, CUDA[18] framework has been chosen to perform the parallel computations.

The desired code would have been packegable as a Python module to allow distribution and reuse of the code. With this idea, the API of the module should have been as simple as possible, giving to the module the hard work of handling complex phases of the overall method.

Lastly, the complete module should produce results compatible with the Matlab code published by [3], to allow those who already use that implementation to switch seamlessly to the parallel high performance version.

This chapter begins with an overview of the world of computing on GPU, followed by the review of some software tools used in this work. After that, the development history of the produced software is described. Finally, compatibility and performance tests results are exposed.

3.1 GPU computing with CUDA

Last decade have seen a tremendous increase of GPUs performance. As the name say, Graphics Processing Units arise from the demand of specialized chips to perform graphics computation. However, since the advent of frameworks like CUDA and OpenCL, GPU could be used for general purpose computing, speeding up computations of scientific research, engineering, medicine and other fields. The power of GPUs comes from the massively parallel computing that can be performed on these devices.

We will go into details of the CUDA framework, that is used in this work to efficiently parallelize the scattering network algorithm.

In CUDA, the parallel portion of the application is called *kernel*. Multiple instances of a *kernel* can be executed concurrently and each instance is called a *thread*. *Threads* are organized in *blocks*, a 1- 2- or 3-dimensional group of *threads*. *Blocks* are also organized in a *grid*. Each level of this hierarchy is executed on the respective hardware level. A CUDA-capable GPU can execute one or more *grids* of *blocks*. On the GPU, many *Streaming Multiprocessors* (SMs) handle the execution of the *blocks*. SMs are composed of many *CUDA cores*, the base ALU of a GPU. Each *thread* is run on a *CUDA cores*.

Therefore parallelization happens at *thread* level. Each *thread* run the same portion of code, but can elaborate different data. For example, let's consider the element-wise sum of two vectors on N elements each. A 1-D *block* of N *threads* can be launched and every *thread* will add together one couple of elements of the vector. This kind of parallelization is called SIMT (Single Instruction Multiple Threads),

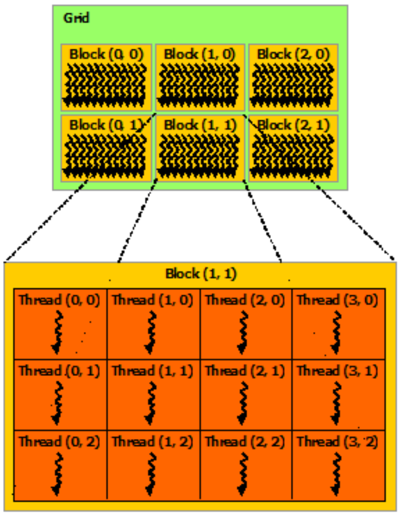


Figure 3.1: Grid of thread blocks. From the Cuda Programming Guide [19].

Multiple operations on a set of data can be chained by concatenating many kernels. If flow of input data is continuous, it can be considered a *stream* and all kernels can run in parallel to perform operations on every item of the stream. This approach of performing parallelization is called *stream processing* and allows to achieve simple form of parallelization.

Parallel computation can be executed from a serial code running on the CPU through the CUDA API.

It's worth pointing out some downsides of computing on GPU. First of all, CPU and GPU usually use different hardware memories, therefore memory copy must be performed at the beginning of the computation to move data from RAM to GPU memory and at the end to get results back to RAM. This copy can take a non negligible time, thus computing on GPU is profitable only if the speedup obtained by the parallelization is enough to ignore the transfer time. A second important point is that GPU architecture have an intrinsic high latency to start computing a thread but can achieve higher throughput compared to CPU. A common solution to both this problem is an efficient application of the *stream processing* paradigm to overlap memory copies and computations.

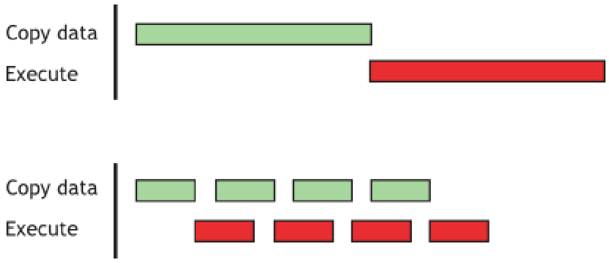


Figure 3.2: A sequential approach (top) and the overlapping of copy of data and thread execution (bottom). Latter approach can speedup execution. From the Cuda C Best Practices Guide [20].

Standard CUDA API are available in C, but bindings and wrappers for many different languages are available.

3.2 Software Tools

Python ecosystem is rich of modules for almost any application. Many of them present core functions written in a compiled language and offer a Python interface to those functions. This method allows to achieve high computational efficiency despite using a simple and flexible interpreted language.

In this work, high performance have been looked for in four main modules: Numpy, PyCuda, Scikit-Cuda and OpenCV.

3.2.1 *Numpy*

Numpy [21] is a Python package for scientific computing that expose a powerful N-dimensional array object that can be used to perform most common and advanced mathematical operations between arrays. The core of the module is written in C to achieve high performance while exposing a simple interface in Python.

3.2.2 *PyCuda*

PyCuda [22] package gives access to Nvidia's CUDA [18] API. The package can be used in two primary ways.

The first is to use the GPUArray[1] object that exposes an interface like Numpy's "ndarray" but stores and compute data on the GPU. Most common methods of Numpy are reimplemented in GPU code to take advantage of parallel architectures. For example, all point-wise operations (e.g. abs, sin, cos, ...) and simple scans/reductions (e.g. min, max, sum, ...) are readily available. PyCuda will handle seamless all operations required by the CUDA API, like instantiating the context, moving data from RAM to GPU memory, creating and running CUDA kernels and getting back results to CPU memory. Instant benefits are usually obtained for large arrays.

The other way to use the module is to write custom CUDA kernels. This approach is suggested to developers that already know how to write own code for GPU but that wants to use a convenient and fast framework that handles all the background work and that allows to concentrate on the actual computing code. In particular, GPUArray's ElementWiseKernel object allows to create a kernel that runs a user-defined operation on every element of the array. PyCuda will handle automatically the JIT (just-in-time) compilation of the kernel, its launch with suitable grid-size and block-size and eventual for-loops within the kernel around the code to achieve optimal performance. At necessity, all CUDA's driver API are available to perform more sophisticated tasks.

3.2.3 *Scikit-Cuda*

Scikit-cuda[23] is a wrapper to many functions of CUDA, cuBLAS, cuFFT and cuSOLVER libraries. It allows to run functions defined in those libraries directly in Python with the help of PyCuda to handle the movement of data to/from the GPU. In this work it has been used primarily for the cuFFT binding. cuFFT [2] is a library that provide an interface to compute FFTs on NVIDIA GPUs. Because the FFT is in operation that can be greatly parallelized, GPUs can yield to important benefits in terms of execution time.

[1] `https://documen.tician.de/pycuda/array.html` Accessed on 11/14/2016.

[2] `https://developer.nvidia.com/cufft` Accessed on 11/14/2016.

3.2.4 *OpenCV*

The most famous library for computer vision is OpenCV [24]. Its Python bindings has been used in first revisions of the code to perform mainly the symmetric padding. This operation was seen to be faster with this library than the Numpy implementation.

During the development, though, the latest version of the library (v 3.1.0) was used to handle images from the loading, saving, resizing and color converting.

3.3 Development

The GPU implementation was developed by writing a prototype of the software and iteratively enhancing it by fine-tuning slowest parts. In the history of this process, three main revisions of the software are reported below to disclose encountered problems and solutions found.

3.3.1 *First implementation*

The first implementation of the code was based on a quasi-straightforward conversion from Matlab code to Python code with the use of GPU capabilities only on the critical step of the convolution in the Fourier space.

The algorithm worked layer-wise, iterating over each layer m to compute new coefficients $S_J[p]x$ and $U[p]x$.

It was clear that almost always at each layer there were multiple signals $U[p]$ with the same size (possibly reduced from the original size by the subsample steps). Since cuFFT implementation is known to work good in batches of signals of same shape[3], this feature was exploited.

At each layer, the algorithm searches for paths whose size is the same and it groups them. Next, each group of signals is elaborated sequentially. Every signal in the group is padded and then they are stacked in a 3-dimensional Numpy array that is copied to GPU in a GPUArray. This array is fed into the cuFFT's FFT routine to compute the transformation in batch. Then the history of every signal is checked and they are multiplied by the appropriate filters, saving results in a new 3-d GPUArray. This array of filtered spectrums is inverse transformed still in batch to obtain wavelet transformed signals. Always on the GPU, the module operation is performed element-wise. Finally, data are copied back from GPU to CPU memory where unpadding and downsample operations were performed.

Performance of this first version were not great, with a very little speedup compared to Matlab code. Most of the time was spent transferring data to and from the GPU and launching kernels. Indeed, GPU functions need to have enough computational work to

[3] `http://docs.nvidia.com/cuda/cufft/#accuracy-and-performance` Accessed on 11/14/2016.

execute so that the overhead time, required by the operation of launching computation and the time-extensive memory copy, is negligible with respect to the actual computation time. The proposed algorithm requires plenty of memory movements so the GPU capabilities are not used at the maximum of their possibilities. This is due to the fact that there were not enough signals of the same shape in each layer to elaborate together.

Nevertheless, this implementation did not require any knowledge of GPU programming, because the only encounter with GPU-related functions is the necessity to call functions to move memory to and from the GPU. Thanks to the Python modules used, this is operation was very easy and others functions ran on the GPU had same syntax of ones that would have ran on the CPU.

3.3.2 *Second Revision*

The second revision of the code took advantage of the observation that the scattering network graph did not need to be forcingly walked layer by layer, but that each path can go on alone, with the only necessity to have completed all paths at the end of the execution.

With this consideration, it can be seen that graph could be walked in such a way that all intermediate signals with the same size are computed at the same iteration. This is accomplished by performing scattering convolution always on those image that have the bigger shape, that are those with the smallest overall downsample rate. If the graph is represented such that filters are scattered left to right according to the increasing j, it's clear that the graph is walked "diagonally", from top left to bottom right.

The implemented algorithm creates and keep a queue of intermediate signals, i.e. incomplete paths. It iterates until there are no more elements in the list by first searching the maximum shape of signals in the list and then grouping all signals with that shape. Those signals are also removed from the list. Then the appropriate filter size for this shape is selected from pre-calculated filters and signals are symmetrically padded and moved to the GPU in stack as in the previous version of the code. The stack is then transformed to the Fourier space and after that each spectrum is point-wise multiplied with appropriate filters. Like before, signals are transformed back in the spatial space with the batched IFFT and the modulo is applied point-wise. The next step is to copy back filtered signals to CPU, where unpadding and downsample is applied. For each signal in the group, if it ended its path it is copied to an output stack, otherwise the intermediate output is added back to the queue list, and another iteration is performed.

As they are usually not needed for subsequent analysis, this approach doesn't save intermediate results $U[p]$.

This evolution of the algorithm allowed to obtain a first interesting speedup relative to CPU implementation. This is achieved by the more effective computation of the FFT and IFFT, that are now executed with the largest possible batches of signals of the same shape. Some memory problems arose because with some combinations of J, L and original image shape, in the firsts steps of the processing the memory required for the IFFT was too large to fit in the not-so-big memory of GPU, especially low-end ones. To mitigate this problem, a solution was found by limiting the size of the batch of signals simultaneously computed: for each group, if its total number of pixels exceed a defined threshold, the group was splitted. This allows to use most of the available GPU memory without having fails during allocation.

Now that the convolution heavy computation is totally delegated to the GPU, the main bottleneck of the process were other operations ran on the CPU, together with the need to move memory to and from GPU too frequently. In particular, it was seen from profiling the code that the padding, unpadding and downsample steps computed on the CPU took about 20 % of the overall execution time.

3.3.3 *Latest release*

The further and last evolution of the algorithm solved bottlenecks of the last revision and achieved highest performance.

This version is based on the good approach of crossing the scattering network diagonally, but it also delegate to the GPU all signals computations by creating custom CUDA kernels and managing efficiently the memory allocation and copy.

In details, the concept of "signals container" is introduced: this is a Python object that contains a GPUArray who store a stack of signals with the same shape (like the concept of groups in previous revision). Moreover, the shape of the GPUArray is such that it can store also the correct padding that is required to handle border problems of convolution. Overall, the GPUArray is 3-dimensional, with a shape given by (B, w_p, h_p) where B is the number of signals that will be stored, h_p and w_p the height and width of the padded version of the signals (Figure 3.3).

A set of containers is created at initialization time, one for each downsample scale that will be met during computation of the scattering network (basing on hyper-parameters M, J, L and image shape). An additional container is prepended to the set to store the original image, that is immediately copied to GPU.

The algorithm basically iterates over the set of containers.

For each containers it pads the image with a custom CUDA kernel created with the "ElementwiseKernel" class offered by "PyCuda", so it is run element-wise over all pixels of signals and padding in the containers. For each pixel, it finds the location on the original image coordinates: if the current pixel is already one of the original image, it is copied in-place; otherwise, if it is one of the border, the corresponding pixel mirrored on the border is copied. This method allows to efficiently and parallely apply the symmetric padding directly on the GPU, without requiring any copy to CPU. No already existing library with this functionality was found, so in previous versions of the code it was required to move image back on CPU also to perform this operation with OpenCV's `copyMakeBorder`.

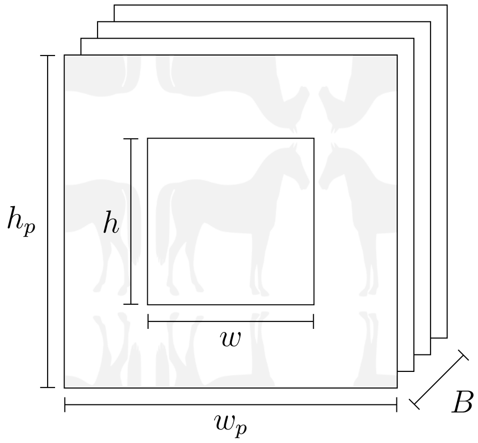


Figure 3.3: Illustration of a "signals container". A stack of B signals is saved in a 3D GPUArray. The signal size is $w \times h$, but a larger space $w_p \times h_p$ is allocated to hold the symmetric padding.

Next, all signals in the container are transformed to the Fourier space in batch. Filters are then loaded, and another custom CUDA kernel is called on each filter to be multiplied for required signals. This allows to load pixels of each filter in GPU registers only once for all the signals that requires that convolution.

Then the IFFT function is called always in batch to transform back signals.

The latest custom CUDA kernel handles operations of unpad, downsample and application of the modulus. This phase is preceded by a preparation step where it is created a list of associations between each convolved signal, related downsample factor and destination container. Then, for each destination container, the custom CUDA kernel is run. Basing on information related to source and destination shape, source and destination padding and downsample, it maps every source pixel to appropriate destination pixel while applying modulo operation. Therefore the computation of the modulo is performed only on useful pixels, while others are discarded.

When the iteration on the last container is concluded, its data (that at this point are scattering coefficients $S_J[p]$) are copied back to the CPU memory and computation is done.

Another feature of this implementation is that allocated GPU memory for containers can be reused for subsequent transformations through the scattering network, so next runs will not have to wait the time required for the memory allocation.

The realization of this approach definitely required CUDA programming skills and knowledge on GPU coding best practices. Custom CUDA kernels are built exactly to optimize the scattering network algorithm, so the implementation is ad-hoc. Previously cited Python modules are used anyway for complex tasks, like the FFT/IFFT transform and the possibility to create effective and simple element-wise kernels. Without those modules, the time required to write the code and its length and complexity would have been certainly higher.

3.4 Packaging and distribution

Last version of the software was packaged as the Python module `scatnetgpu`.

The module have several dependencies:

- `numpy>=1.11.0` Python library for CPU numeric computations
- `scipy>=0.18.0` Python library used only to calculate the binomial coefficient
- `pycuda>=2016.1.2` Python library for heavy computations on the GPU
- `scikit-cuda>=0.5.1` Python library for the cuFFT wrapping
- `octave` framework, `oct2py` Python library, *Scatnet Matlab code* [4] required to load the filter creation function of the original Matlab code through *octave* (necessary as long as that part of the code is not ported to Python)

The main class available in the module is `ScatNet`. The class can be instantiated by passing parameters M, J and L of the required scattering network. The resulting object have the useful method `transform` that perform the scattering network transformation of a given image (represented by a Numpy `ndarray`). This method takes care of creating filters if they are not already present in cache, then it performs the transformation and returns the result in form similar to the one of the Matlab code. If a batch of images needs to be transformed, the method `batch_transform` takes care of looping over the batch. If given `ndarray` have 3-dimensions, the last one is considered to represent channels (e.g. RGB, HSV, ...). Every channel is transformed alone, and resulting representations are concatenated.

An additional utility function, `stack_scat_output`, takes the transformed representation and returns a 3-dimensional `ndarray` where each scattering output is stacked in the first dimension, in addition to a list of meta-data to recognize the paths of the network. This was useful for the later classification procedure, because the order of the output is not important in this task.

The source code has been released open-source under MIT License and it's available at: `https://github.com/oinegue/scatnegpu`. From the link, the code can be reviewed, downloaded, easily installed and modified. In this way, I hope to make available to other researchers a fast (as further discussed in Section 3.6) implementation of the scattering network algorithm, in order to continue studies for new uses of this representation and to permit the realization of real-time applications.

[4] `https://github.com/scatnet/scatnet` Accessed on 11/19/2016

3.5 Compatibility Tests

The GPU implementation was tested to be compatible with the reference Matlab code. Tests were performed at many levels of aggregation, to check that any combination of parameters leaded to compatible scattering coefficients in output.

Firstly, outputs of the GPU code and the Matlab code were compared pixel-by-pixel. We are looking at the result of a machine computation, thus the analysis must take in account that number are expressed in a floating point representation, as expressed in the standard [25]. In this specific case, all computations are performed in single precision, so each value is stored in memory with 32 bits.

The absolute difference $|x_1 - x_2|$ between two machine numbers x_1 and x_2 can show qualitatively if the results are nearly correct. To quantify the magnitude of the eventual difference, it's important to validate discrepancies of the results with respect to the capabilities of the single precision floating point representation used.

A generic approach to this problem is to use the concept of *Unit in Last Place* (ULP) [26]. Given a floating point number x whose exponent is E, it is defined $ULP(x) = (0.0...01)_2 \times 2^E$. More intuitively, the ULP is the gap between x and its very next larger floating point number. Therefore a method [27] to evaluate the difference between two floating point numbers is to count how many ULPs there are between them. This count is usually called NULP (Number of ULPs).

Figure 3.4 represents an example of the tests. It can be observed that both version appears qualitatively similar in first two columns. Nevertheless, the absolute difference is non zero almost everywhere. This may be due to the different implementations of functions like the FFT/IFFT, to round-off error of the single precision floating point representation used and to its propagation with the particular order of the operations performed in the final revision of the algorithm. However, the NULP difference exposed in the last column is limited. Considering the big number of operations involved in the scattering network base algorithm, this result is considered acceptable. Therefore, standing to this first result, the two algorithms are compatible.

The study followed by analyzing the biggest NULP difference in the output signals of each path of a scattering network. Box plot of Figure 3.5 is built by computing the maximum NULP difference of each path of a scattering network and collecting this result for 100 different input images from the textures dataset used in Chapter 5. It's clear that even with a large set of images, the NULP difference between the two algorithms remains low. Some outlier values reach higher difference. It was observed that these discrepancies are higher in regions of images that are nearly black. This may be due to round-off errors that for small numbers are amplified during the FFT/IFFT.

Compatibility has been tested over a range of different image sizes and scattering network configurations, to check if the implementation was correct. The Figure 3.6 show one of the tested scattering network configuration. It can be observed that the maximum

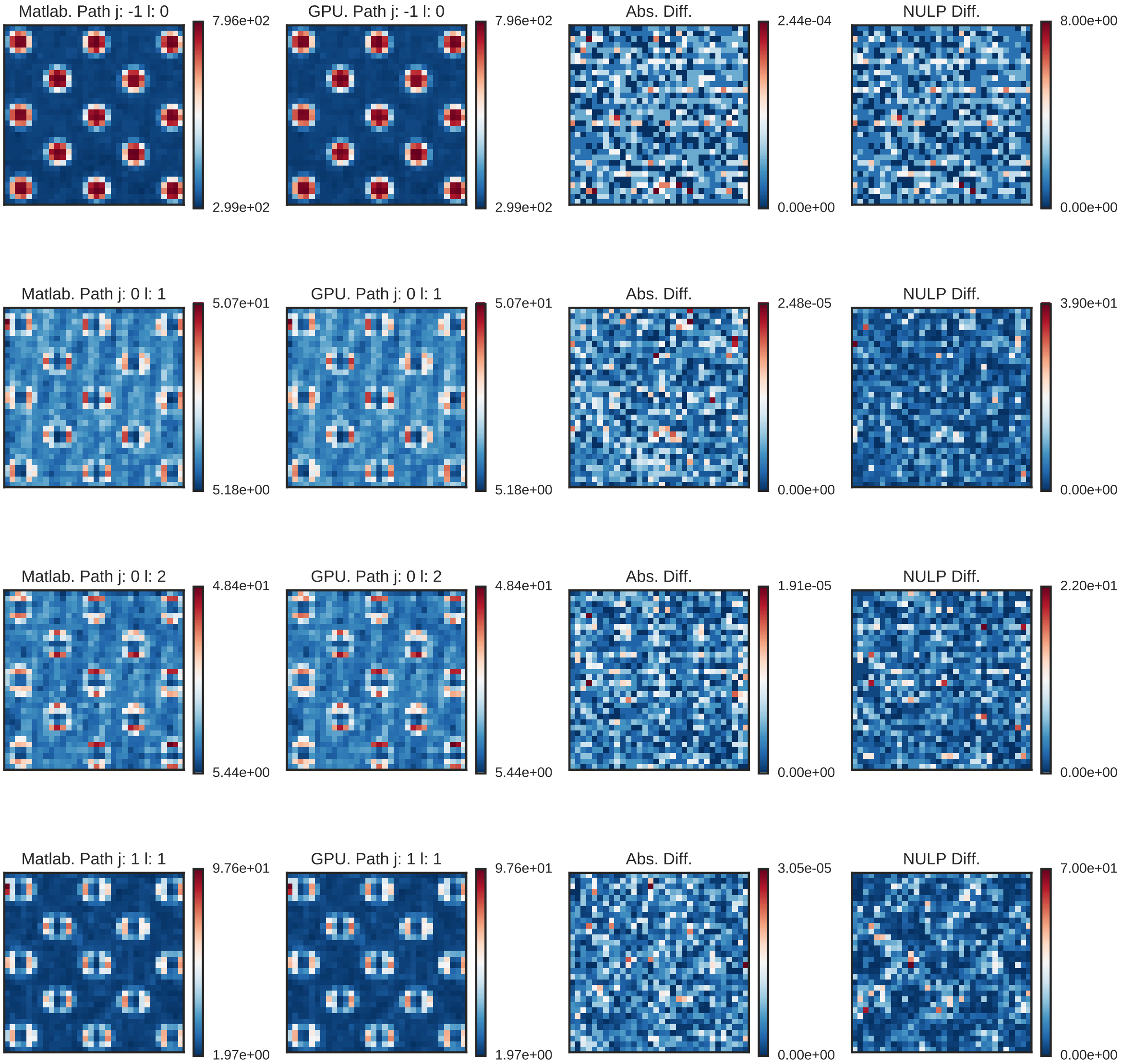


Figure 3.4: Representation of the compatibility of the outputs for a polka dot image. Original image size was 128 px × 128 px. Scattering network parameters were $M = 2$, $J = 2$, $L = 2$. From left to right: output of Matlab code, output of GPU code, pixel-wise absolute difference, NULP difference. From top to bottom, different paths of the scattering network are shown.

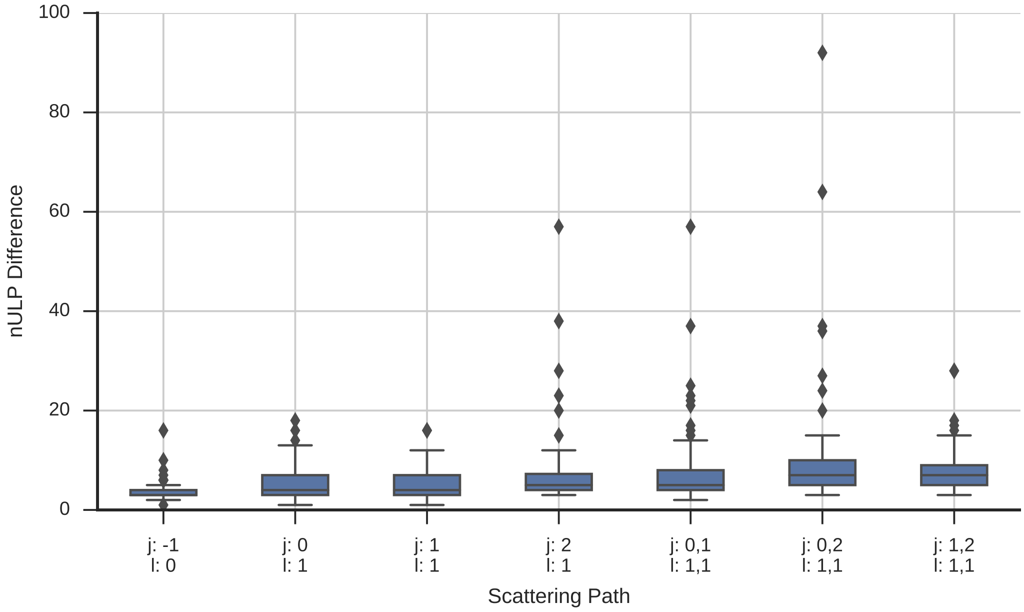


Figure 3.5: Maximum NULP difference between Matlab and GPU implementation for each scattering path of a network with $J = 3$ and $L = 1$. 100 repetitions were run with different images.

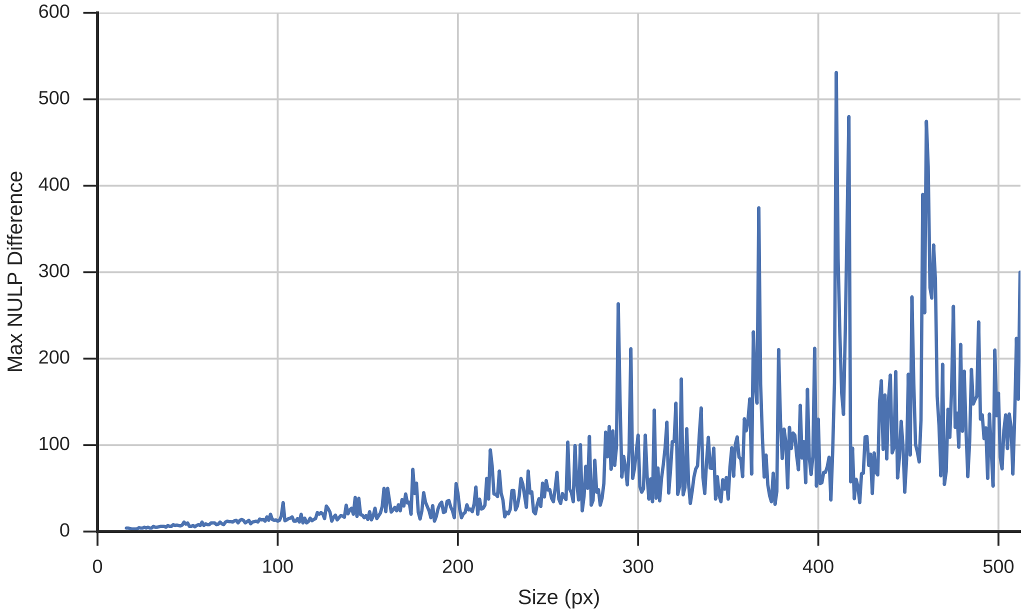


Figure 3.6: Maximum NULP difference between Matlab and GPU implementation for increasing image size with $J = 3$ and $L = 1$. 10 repetitions were run for each size. Median value is displayed as a solid line.

NULP difference increases with the image size. It's hard to define an objective threshold for the compatibility of the results of a floating point calculation. Nevertheless, most of the values stay below 500 NULP of difference. This value seems to be a good result compared to the 2^{24} values that are representable with the same exponent of a single precision floating-point number.

If further accuracy is needed, the causes of the incongruity should be searched in every function used in the algorithm, by comparing Matlab results with Python/CUDA results. Moreover, the particular order of functions in the algorithm could lead to different round-off errors that accumulates to create the different results.

3.6 Performance

Execution times of the scattering network algorithm were compared for both the GPU implementation and the Matlab code running on the CPU. Tests were performed on the following machines:

lenome **notebook** A personal notebook with a discrete graphic card

- Model: Lenovo Flex 2-14
- CPU: Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz
- RAM: DDR3 1600 MHz 8 GB
- GPU: Nvidia GeForce 840M
  - Architecture: Maxwell
  - CUDA cores: 384 CUDA
  - Max Clock: 1.12 GHz
  - Memory: DDR3 4 GB
- OS: Ubuntu 15.10
- CUDA version: 7.0
- Matlab version: $R2016b$
- Octave version: 4.0.0

phantom **desktop** An assembled desktop used for basic machine learning tasks

- CPU: AMD FX(tm)-8350 Eight-Core Processor
- RAM: DDR3 1866 MHz 16 GB
- GPU: Nvidia GeForce GTX 960
  - Architecture: Maxwell

  - CUDA cores: 1024 CUDA
  - Max Clock: 1.29 GHz
  - Memory: GDDR5 2 GB
- OS: CentOS 7.2.1511
- CUDA version: 7.5

***titano* workstation** An assembled workstation used for heavy machine learning tasks

- CPU: Intel(R) Core(TM) i7-5930K CPU @ 3.50GHz
- RAM: DDR4 2133 MHz 32 GB
- GPU: Nvidia GeForce GTX TITAN X
  - Architecture: Maxwell
  - CUDA cores: 3072 CUDA
  - Max Clock: 1.08 GHz
  - Memory: GDDR5 12 GB
- OS: Ubuntu 14.04.5
- CUDA version: 7.5

First of all, the execution time of the mere transformation function was measured (`ScatNet.transform` for Python, `scat` for Matlab). Test was performed by running the function in loop for 10 executions and measuring the mean time required for each execution. The measuring was repeated 5 times and best result was reported to mitigate possible breaking due to the operating system. Several combination of image size S, filters scales J and rotations L were tried (an image of size S is intended to be a gray-scale square image of size Spx $\times$ Spx).

In Python, time was measured with the `time.clock()` function [5]. In Matlab, the couple of functions `tic; <code>; tac;` was used [6].

Figure 3.7 and Figure 3.8 reports measured performance of two extremities of parameters combinations, respectively a small network with $J = 3, L = 4$ and a large network with $J = 5, L = 8$. Different image size were tried, from $S = 32$ to $S = 1024$ at steps of $\Delta S = 32$. In both cases, the same overall trends are observed: GPU code is faster than CPU code, more powerful GPUs scores faster results, Octave interpreter is slower than Matlab interpreter.

Starting the analysis from the last point, the speed-up of Matlab with respect to Octave is up to 4 times for small images and for both parameters configurations. Speedup

[5] `https://docs.python.org/2/library/time.html#time.clock`. Accessed on 11/18/2016.

[6] `https://it.mathworks.com/help/matlab/matlab_prog/measure-performance-of-your-program.html`. Accessed on 11/18/2016.

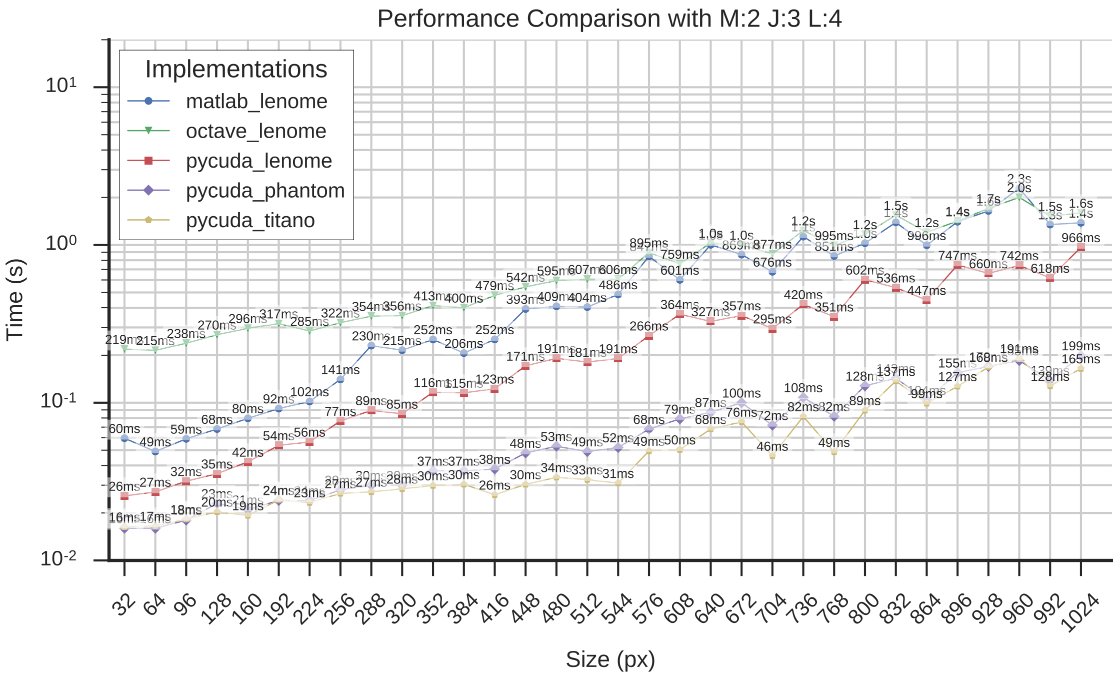


Figure 3.7: Execution time of the scattering network transform with $M = 2$, $J = 3$ and $L = 4$ at different image size S.

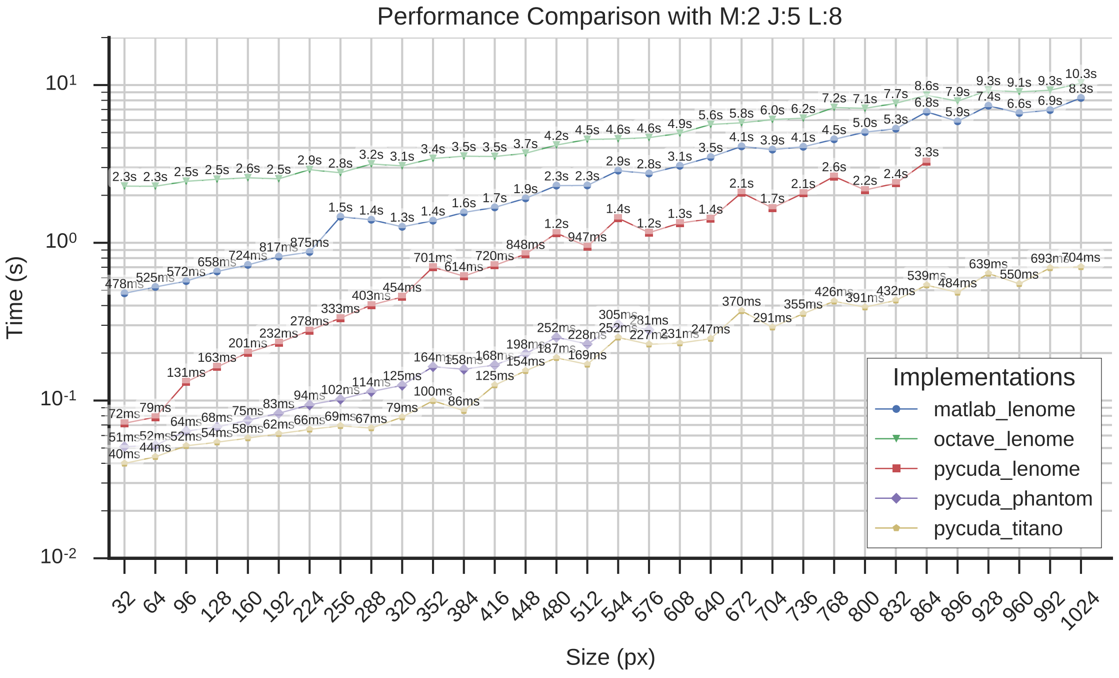


Figure 3.8: Execution time of the scattering network transform with $M = 2$, $J = 5$ and $L = 8$ at different image size S.

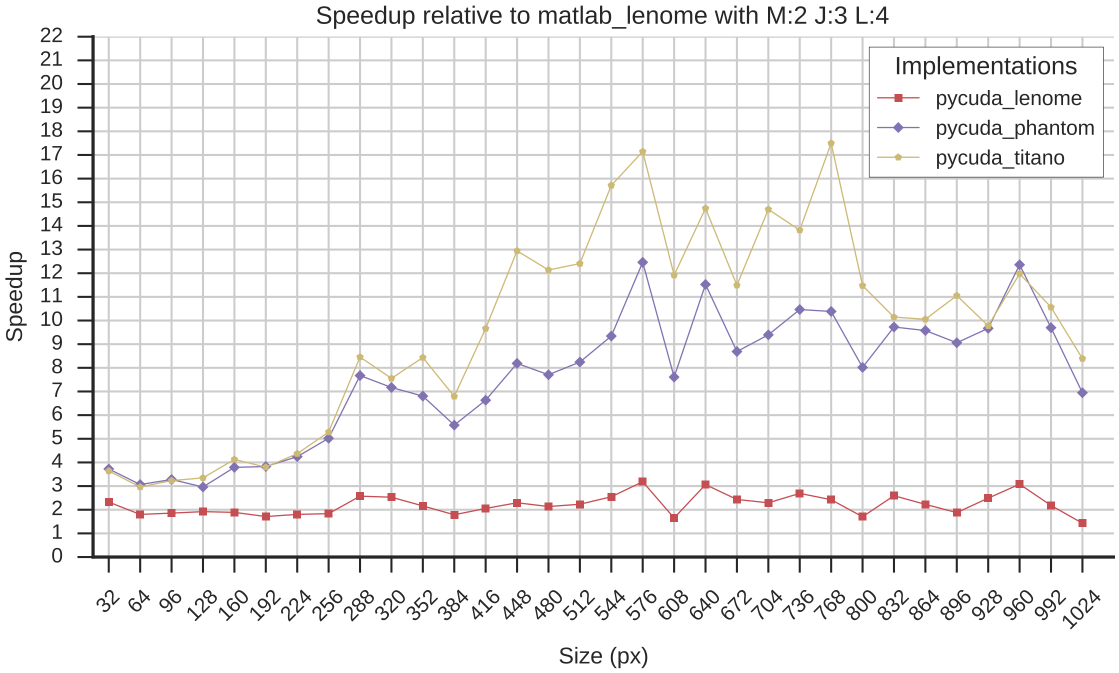


Figure 3.9: Speedup of the GPU implementation with respect to Matlab version for the scattering network of Figure 3.7

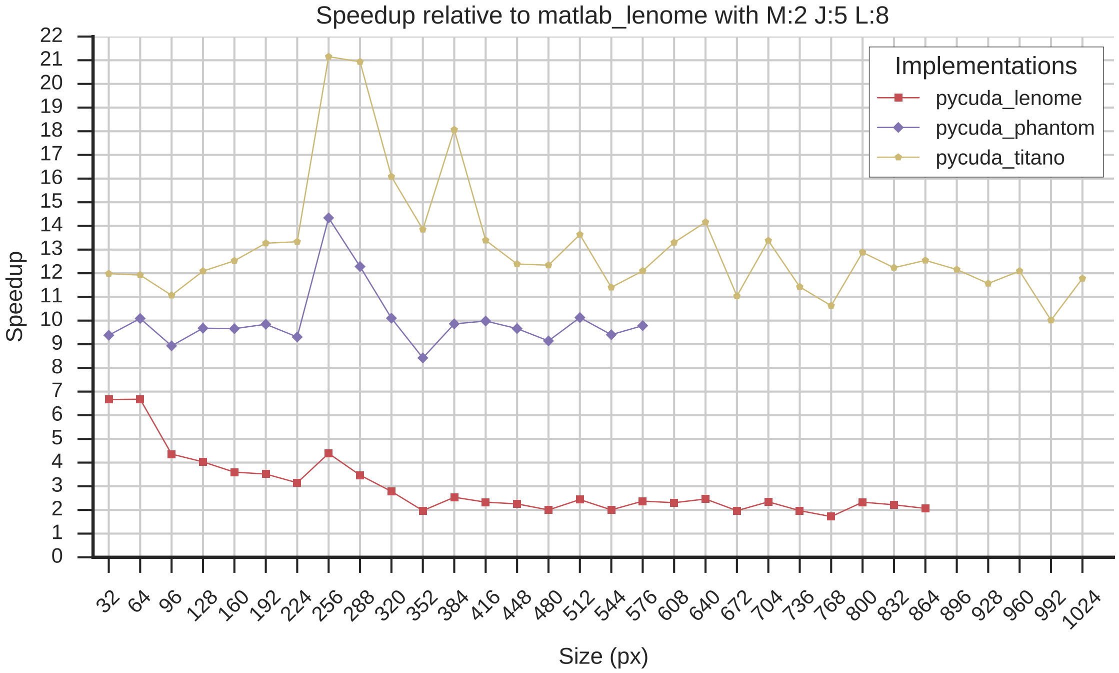


Figure 3.10: Speedup of the GPU implementation with respect to Matlab version for the scattering network of Figure 3.8

drops for big images, scoring similar results for the small network and a little speedup for the large network. Better performance may be due to many optimizations of the Matlab environment not present in Octave. For example, it was observed that many cores were busy during Matlab execution while Octave run occupied only one core. This could mean that in Matlab some functions used in the algorithm are parallelized (like the FFT or matrix multiplications). A further study of the differences between the two environments is beyond the scope of this work and then, because of the higher performance, subsequent comparison will be based only on time measures in the Matlab environment.

Performance improvements of the GPU implementation relative to Matlab code depends obviously on the specific GPU. In Figures 3.9 and 3.10 it can be seen that for the entry-level GPU of *lenome* notebook, the speedup is around 2× for the small network and peaks at more than 6× for the large one and small images, decreasing again to 2× for bigger images.

Medium range GPU of *phantom* desktop have better performance. The small network configuration shows a speedup from 3× up to more than 10×. Speedup increases with image size, showing that the potential parallelism of the GPU is exploited when occupancy of the cores is high. The big network shows a mean performance increase of about 10×, with a peak when the Matlab code show an important slow-down at around $S = 256\,\text{px}$.

Top-level GPU of *titano* scores top performance with a general trend similar to that of the previous graphic card, but with an additional speedup relative to the Matlab code, up to 17× for the small network and 21× for the bigger one. Again, the computational capabilities of these devices is not completely accessible for low complexity tasks (small network and image sizes), but the performance gain is important for heavy calculations.

Looking at absolute values, the new GPU implementation can reach nearly real-time performance in several combinations of parameters. For big networks, mid/high GPUs scored less than 100 ms (10 FPS) for images up to 256 px. This result can be useful for practical applications where a high rate of processed images is needed, like in industrial environments or video processing.

All curves presents spikes at certain image sizes. They are due to the performance of the FFT and IFFT algorithm at different signal sizes. In fact, standing to the documentation of cuFFT[7], the implementation is optimized for image sizes that can be factorized as powers of small prime numbers, i.e. $S = 2^a 3^b 5^c 7^d$. If the size can not be factorized in this way, more computation are required and execution time increases. Figure 3.11 shows the described behaviour: time required to run an FFT varies greatly even changing image size of only 1 px. Due to the padding, this situation can happen despite the initial image size. A better implementation of the algorithm could estimate the optimal padding for this requirement. This optimization has not been implemented in the last revision of the software because filters creation is still based on the original Matlab code.

[7] `http://docs.nvidia.com/cuda/cufft/#accuracy-and-performance` Accessed on 11/18/2016.

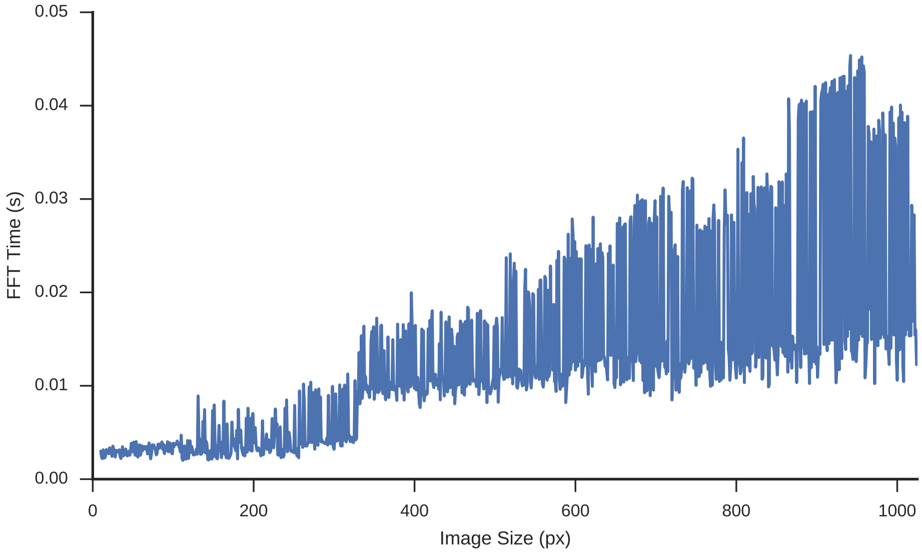


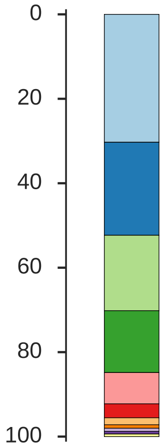
Figure 3.11: Time required to run a single 2D FFT transform on a square gray-scale image with cuFFT through Python bindings offered by Scikit-Cuda (averaged over 10 runs).

A final note is that many points are missing in the big network chart, at large image sizes, in the GPU curves of *lenome* and *phantom*. Memory required to perform the (I)FFT depends heavily on the image size and for some "unlucky" dimensions it could be as large as 8 times the memory of the original signal [8]. The (I)FFT in the algorithm is run in batches, so the memory requirement may be too large to be allocated on the limited memory of the GPU and so execution will fail for this sizes. Two methods are possible to bypass the problem. The first is to limit the number of signals in the batch and instead run multiple batches of (I)FFT. The second method is to split the image in multiple patches, perform the scattering network transform on each patch and finally recombine the outputs. The latter approach generalize better for very large images, but requires attention to treat correctly the behaviours at borders.

Every code enhancement was guided by the time profiling of the previous version. Each macro block of the algorithm was enclosed in a couple of stopwatch timers to measure how much time was spent in each part of the code. In the following, the time profiling of executions of the latest code on *phantom* is shown.

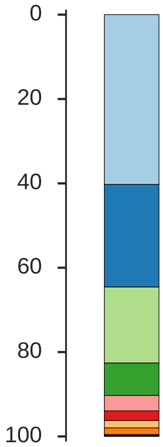
Table 3.1 reports the time profiling of a run of the small network of Figure 3.7 launched on *phantom* with image size 512 px. First observation is that more than half of the time is used for the entire convolution operation (FFT, Element-wise Multiplication, IFFT), that involves 67 % of the total computation. Surprisingly, the copy of the filters

[8] `http://docs.nvidia.com/cuda/cufft/#cufft-setup`. Accessed on 11/18/2016.



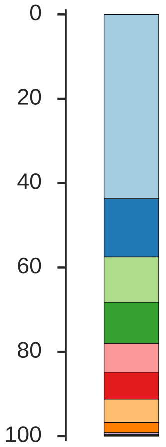
Color	Operation	Percentage	Time
🟦	IFFT	30%	20.8ms
🟦	FFT	22%	15.1ms
🟩	Filters copy CPU → GPU	18%	12.3ms
🟩	Element-wise Multiplication	15%	10.1ms
🟥	Unpad / Downsample / Modulus	7%	5.1ms
🟥	Symmetric Padding	3%	2.3ms
🟧	Containers Allocation	2%	1.1ms
🟧	Preparation	1%	0.6ms
🟪	Results copy GPU → CPU	1%	0.5ms
🟪	Image copy CPU → GPU	1%	0.4ms
🟨	Format Output Data	1%	0.4ms

Table 3.1: Time profiling of the Scattering Network algorithm on *phantom*'s GPU. Image size was 512 px × 512 px, parameters were $M = 2$, $J = 3$, $L = 4$. Number of scattering coefficients produced is 249856 Times averaged over 100 runs.



Color	Operation	Percentage	Time
🟦	IFFT	40%	100.0ms
🟦	Filters copy CPU → GPU	24%	60.5ms
🟩	Element-wise Multiplication	18%	44.8ms
🟩	FFT	8%	19.1ms
🟥	Unpad / Downsample / Modulus	4%	9.1ms
🟥	Preparation	2%	5.7ms
🟧	Containers Allocation	2%	4.3ms
🟧	Format Output Data	2%	4.0ms
🟪	Image copy CPU → GPU	0%	0.5ms
🟪	Results copy GPU → CPU	0%	0.4ms
🟨	Symmetric Padding	0%	0.2ms

Table 3.2: Time profiling of the Scattering Network algorithm on *phantom*'s GPU. Image size was 512 px × 512 px, parameters were $M = 2$, $J = 5$, $L = 8$. Number of scattering coefficients produced is 174336. Times averaged over 100 runs.



Color	Operation	Percentage	Time
🟦	Element-wise Multiplication	44%	27.1ms
🟦	IFFT	14%	8.5ms
🟩	FFT	11%	6.7ms
🟩	Preparation	10%	6.0ms
🟥	Format Output Data	7%	4.2ms
🟥	Containers Allocation	6%	4.0ms
🟧	Unpad / Downsample / Modulus	6%	3.4ms
🟧	Filters copy CPU → GPU	2%	1.5ms
🟪	Symmetric Padding	0%	0.2ms
🟪	Image copy CPU → GPU	0%	0.2ms
🟨	Results copy GPU → CPU	0%	0.1ms

Table 3.3: Time profiling of the Scattering Network algorithm on *phantom*'s GPU. Image size was 32 px × 32 px, parameters were $M = 2$, $J = 5$, $L = 8$. Number of scattering coefficients produced is 681. Times averaged over 100 runs.

from CPU to GPU is another important part of the transformation, that spends nearly one fifth of the time moving filters. This is because filters in the Fourier space occupy lot of memory and the memory transfer speed to and from the GPU is still one of the main bottleneck of GPU programming.

It would be interesting to evaluate the possibility to create filters directly on the GPU to avoid this memory copy, but the trade-off with the computation time required for their creation may not worth the hassle.

At the opposite, operations like symmetric padding, unpadding, downsample and modulus benefit a lot of the custom implementation in CUDA kernels. Their overall contribution is about 10 % and hence does not influence essentially the overall timings.

The container allocation operation refers to allocating memory on the GPU device to store intermediate results of the algorithm. If a batch of images needs to be transformed, this operation is required only once because allocated memory would be reused, saving a bit of time.

An analogous operation with the filters copy was not possible because the whole filter bank is too big to be stored on the GPU when other memory intensive operations like FFT/IFFT need to be performed.

Format Output Data represents the operation of organizing results like original Matlab code does and Preparation is a group of operations that arrange the environment for the execution of the algorithm. Finally, transferring input image from the CPU to GPU and getting scattering coefficients back are cheap operations because the size of the data is small.

Increasing network complexity changes marginally some of the percentages. Table

3.2 show the profile of the larger network whose performance were presented on Figure 3.8. In this case, filters transfer to the GPU requires more time because the number of filters is larger in this configuration.

A different behaviour can be seen if the input image size is reduced. Table 3.3 show the time profiling of the transformation of an image of size 32 px × 32 px. In this case filters size is really smaller than before, so filters copy time is far less influential (2 %). It's worth pointing out that the element-wise multiplication is the slower operation in this case. This is due to the fact that a multiplication kernel is launched for each filter, so if signals are small, the overhead of launching the kernel is too big to be overwhelm by the actual computation. This passage has not be efficiently optimized to keep the code more readable, but if better performance are needed a more complex custom CUDA kernel could be wrote to efficiently apply the element-wise multiplication over all filters with a single launch.

CHAPTER 4

Training and Test Methodology of a Machine Learning Problem

Training a machine learning algorithm requires many steps, organized in a severe structure to prevent logic errors. This chapter explains the general workflow of how to train a machine learning algorithm and will point out the particular approach required for the task of classification of images with the scattering network representation.

4.1 Train/test sets and cross-validation

A machine learning algorithm is trained on a set of data, usually called database or dataset. Generally, samples of a database have to be used both for the training and for the evaluation of the algorithm. It would be a methodological mistake to use same data for both tasks. That procedure would cause the "overfitting", a situation in which the trained model can perfectly score on the whole database but would fail miserably when seeing new data.

Therefore the first critical step is to split the database in two sets: a train set and a test set. The train set will be used for all the training procedure, ranging from casual tests, model selection, hyper-parameters search, untill the final training. The test set will be used only at the very ending of the training to evaluate performance of the final trained algorithm. It's important to point out that the samples have to be chosen randomly, so that the two sets will be composed of independent and identically distributed ($i.i.d$) random samples.

In this work the split has been chosen to be in a 70% / 30% proportion, meaning that 7 random samples out of 10 of the whole database has been used for the training, while the remaining 3 have been put apart and will be used only for the test.

After this split, the train set will be used to choose an optimal model. It this task, it is required to have another set to be used for the "validation" of the model. As the number of samples in the provided database is relatively small, a technique called "k-Fold Cross-Validation" have been used. This approach divides the train set in k "folds". Then the model is trained using $k-1$ folds and validated on the remaining fold. The procedure is looped over all combinations of folds and the performance measure is averaged and returned.

Common applications of this approach are with 8 or 10 folds. The extreme case is the "leave-one-out", that consists of having a number of folds equals to the number of samples.

The cross-validation is a computationally expensive method, but it can avoid wasting useful data.

In this work the cross-validation have been done with 8 folds as a compromise between computation time and dimension of the training set.

4.2 Pre-processing

Samples from the database may not be already suitable for the subsequent feature extraction and classification steps. Therefore some kind of pre-processing needs to be done on data to adapt them to the requirements of the learning algorithm.

A frequent requisite for image classification models is to have a fixed input image size. This is due to the fact that the architecture of learned parameters cannot be used with different sizes.

As the output size of the scattering network directly depends on the size of the input image, it was necessary to take all images to the same size. This was accomplished by a three steps resize to bring all different sizes present in the database to a common size. Steps followed are:

- Down-scaling for an integer factor: this step simulate having a camera sensor whose pixels size is an integer multiple of the original pixels size. This leads to an averaging of the signal of adjacent pixel like the one that would be observed in a physical situation.

- Shrink to the required size by keeping aspect ratio, so the maximum dimension of the required size is equal to the minimum dimension of the shrinked image

- Crop to have the required size

This approach allows to treat better the high frequency information that could be discriminative for image classification.

The first two steps have been done with a bicubic interpolation to better preserve edges.

The choice of the optimal size of the resize is not known *a priori* and it is neither a parameter that is learned during the training. Therefore it is called an “hyper-parameter” of the model. This “hyper-parameter” will be chosen by evaluating several possibilities (i.e. sizes of 256×256, 512×512, 1024×1024) with the cross-validation method and finding the one that gives best classification performance.

4.3 Data Augmentation

Machine learning algorithms generalize better the more are the data they are trained on. In the image classification task, an effective technique used in several works is data-augmentation, that consists on creating fake data to add to the training set.

The new fake data are helpful to the classifier to be better invariant to the transformations applied to original data to obtain fake data. An obvious requirement is that this new data should belong to the same distribution as original ones, and this is achieved by choosing optimal transformations for the supposed distribution.

Data augmentation of images is usually performed with small affine transformations or a subset of them.

In particular, in this work a weak data augmentation has been performed by rotating training data at multiples of 90°. In this way the train set increased fourfold without introducing artifacts because these rotations does not need any form of interpolation. This choice was based on the fact that the category of analyzed images did not depend on a particular orientation.

Following the methodological thinking of Section 4.1, data augmentation have been performed only on the samples of the $k-1$ folds used for training in the cross-validation.

4.4 Feature Extraction

The scattering network representation proposed in Chapter 2 is a method of feature extraction: a set of features is obtained by an input data with specific operations. Therefore input data are projected in a new space where they are supposed to be invariant or stable to specific transformations of the original space and, most important, better discriminable.

This procedure is applied after the creation of the augmented training set: every sample is transformed with a scattering network of given parameters “M”, “J” and “L”. These are also “hyper-parameters” that will be selected with cross-validation.

In this work several combinations of $J \in \{6, 7, 8\}$; $L \in \{6, 8\}$ were tried to reach highest classification performance. M was keep fixed to 2 because reference paper [3] stated that the classification metrics did not really improve with more than 2 layers of scattering, despite the extreme increase of the time required to apply the transformation.

4.5 Feature scaling

Sometimes it is required to transform features to make them more adequate to be an input of the following classifier.

For the SVM classifier, it is always suggested to scale input data to be in the range $[-1, 1]$ or $[0, 1]$. This will speed up convergence and will prevent some possible numerical imprecisions (but this is not a hard requirement).

The common approach to map features to the range $[-1, 1]$ is to subtract the mean over the train set and then divide for the maximum of the absolute values. This operation is performed feature-wise (each feature for itself). Despite the analytically correctness, this method may give some trouble if there are big outliers in the training set: small but possibly important features would be scaled to a very small range and floating point precision may not be enough to express their discriminability.

An alternative method is to suppose features distributed normally around the mean and to scale data by 3 times the standard deviation σ (after having removed the mean μ). With this scaling, about 99.7 % of data would be in the range $[-1, 1]$ and only outliers would remain outside it. Sometimes this approach (with different multiples of σ) is called "standardization".

In this implementation, the mean μ and the standard deviation σ are calculated feature-wise on the features extracted from the augmented training set. Therefore these are two parameters "learned" from the data.

4.6 Classifier training

The crucial step of a learning algorithm is the training of the estimator. In a supervised learning context, this step consist on feeding processed features and linked labels to the learning algorithm. Depending on the algorithm, the training can take a lot of time or be almost instantaneous.

One of the most famous and appreciated learning algorithm for classification is the Support Vector Machine proposed originally by Cortes and Vapnik [28].

4.6.1 *Support Vector Machine*

Support Vector Machine (SVM) is a binary linear classifier. Given a set of training data, a SVM represents samples as points in an N-dimensional space and builds a hyper-plane that can discriminate new data between the two classes if they lay respectively on one side or the other of the hyper-plane. The hyper-plane is chosen such that it creates the widest possible gap between samples of the two classes, defined the hard-margin. The larger is the margin, the better the hyperplane separates data. In the general case where data are not linearly separable, the SVM uses a soft-margin formulation [29, 28], where the hyperplane is allowed to misclassify some points, but a penalty is applied. This is achieved using the so-called *hinge loss*. Overall, the SVM tries to minimize the following optimization problem:

$$\min_{w} \frac{1}{n} \sum_{i=1}^{n} \max\left(0, 1 - y_i(w \cdot x_i)\right) + \frac{\lambda}{2}\|w\|^2$$

Usually, the misclassification cost parameter C is introduced instead of λ to scale the empirical *hinge* loss. Those two are related by $\lambda = \frac{1}{nC}$. This gives the formulation:

$$\min_{w} C \sum_{i=1}^{n} \max\left(0, 1 - y_i(w \cdot x_i)\right) + \frac{1}{2}\|w\|^2$$

In this formulation, the parameter C can be considered a trade-off between a better separation of the training data (large C) or a larger margin of the hyper-plane (small C). Figure 4.1 explains visually this trade-off with a synthetic 2D binary classification problem. Smaller values of C creates a higher margin, but some points are misclassified. When C grows, less points are included in the smaller gap of the soft-margins and thus the number of misclassified points is lower.

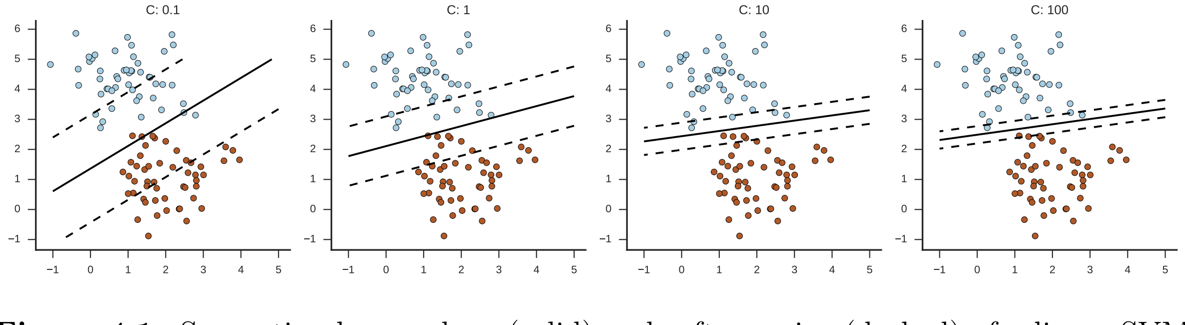


Figure 4.1: Separation hyper-plane (solid) and soft margins (dashed) of a linear SVM at different values of C.

Several ways have been proposed for solving this problem. Most famous are reducing the minimization to a quadratic programming optimization problem, where the primal or dual problem can be solved.

More recent approaches are instead based on sub-gradient descent of the hinge loss or coordinate descent.

SVMs can be generalized to learn non-linear classification by applying the *kernel trick*. This approach uses the observation that during the optimization all input data are used for computing pairwise inner products $< x, x' >$ and that a transformation of the input points $\Phi(x)$ can be broadcasted in the inner product by computing a kernel $k(x, x') =< \Phi(x), \Phi(x') >$. In this way, if a kernel function $k(x, x')$ could be given, the representation $\Phi(x)$ is not really needed to be computed. An example of successful kernel used in conjunction with SVM is the Radial Basis Function (RBF) kernel $K(x, x') = \exp(-\frac{\|x-x'\|^2}{2\sigma^2})$ that implicitly maps input data to a feature space with an infinite number of dimensions.

Despite this possibility offered by SVMs, the kernel trick has not been used in this work. Instead a simple linear SVM was used because dimensionality of the feature space provided by scattering networks is enough large and kernel tricks wouldn't bring in a valuable increase of performance, while increasing computational complexity.

A final note should be done to a generalization of SVMs to deal with multiclassification task. This is usually achieved by training multiple SVM binary classifiers in one of the two configurations: one-vs-one or one-vs-rest. The former approach trains multiple SVMs to discriminate between each pair of classes. Each classifier is than applied to new data and the class with the higher number of winnings in classifications is the one assigned to the data. The latter method trains SVMs to discriminate one class from all the others. The winner class is the one that score the highest output function in it's classifier.

In this work SVM has been used to learn from the augmented train set represented with the scattering network and scaled. The software implementation used in this work offered some hyper-parameters to tune the model. In particular, the cost C has been tried in the range from 10^{-6} to 10.

4.7 Implementation and practical details

The training and evaluation procedure described above had been followed with the help of some Python scripts to automate some steps.

Despite the order of the operations described above, some steps that didn't depend directly on the train set were performed in batch before the actual cross-validation. This allowed to speedup the grid-search of hyper-parameters because most of the computation were already done and cached on disk.

The shrinking at different sizes was applied over the whole database with the software

Imagemagick[1] ran with GNU parallel [30] to use all cores available on the machine.

Data augmentation was performed *on the fly* while extracting features with the scattering network. In details, a Python script loaded each shrinked image and applied the 3 rotations to obtain 4 images. Each channel of each rotated image was supplied to the scattering network to obtain a feature vector of the scattering coefficients. Feature vector of each channel of a single image were then concatenated and stored on disk with an association to the original image, to the shrinking applied, to the rotation performed and to the scattering network parameters used. This allowed to apply a scattering network with the same parameters to a big batch of images all of the same size, improving computation times as described in Subsection 3.3.3.

In this way, the script that performed cross-validation didn't really computed the data-augmentation and the scattering network, but simply loaded right feature vectors from the disk. This technique allowed to avoid time wasteful multiple computations of the same transformations.

The cross-validation procedure was based on the `KFold` class of Scikit-learn [2] [31].

The LIBLINEAR [32] library, in the multi-core implementation [33, 34], was used to train the linear SVM model. The library allows to choose the values of several parameters, including optimization problem solver type, misclassification cost C, tolerance of termination criterion *epsilon*, whether to include or not the bias and specific weights to adjust C of different classes. Python binding of the library were helpful to automate the cross-validation procedure.

[1] `www.imagemagick.org`. Accessed on 11/18/2016

[2] `http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html`. Accessed on 11/18/2016

CHAPTER 5

Textures Classification

5.1 Database

In [3], the scattering neural network representation was tested for digits and textures classification. In particular, the textures classification task required to classify images from the CUReT texture database [35]. The database consists in photos of surfaces of 61 real-world samples taken at different poses and illumination conditions. Thus this collection was primarily targeted to studies of the reflactance of different materials.

The use of a 2 layers scattering network, followed by PCA classifier, allowed to obtain high classification results, with a percentage of errors as low as 0.2 %.

Guided by this achievement, a test of the general purpose scattering network representation on a newer database of textures was performed. The database used is the "Describable Textures Dataset" (DTD) [4]. This database is composed of 47 categories and 120 images for each category, for a total of 5640 images. Authors of the database explain that the database was build after a vocabulary of 47 *terms* or categories to describe textures. These *terms* would allow to describe most of the textures and patterns that can be found *in the wild*, i.e. not generated in a controlled environment. Therefore many images of real world objects and scenes were searched and selected to build the database. The result is a very heterogeneous database, where the intra-class variability depends evidently on high level concepts. Some images belonging to a class per row are showed in Figure 5.1. Authors of the database also reported the accuracy performance of classification with state-of-the-art descriptors, encodings and classification methods. Best results are 61.2 % with a shallow description based on SIFT and 66.7 % with a deep representation obtained with convolutional neural networks.

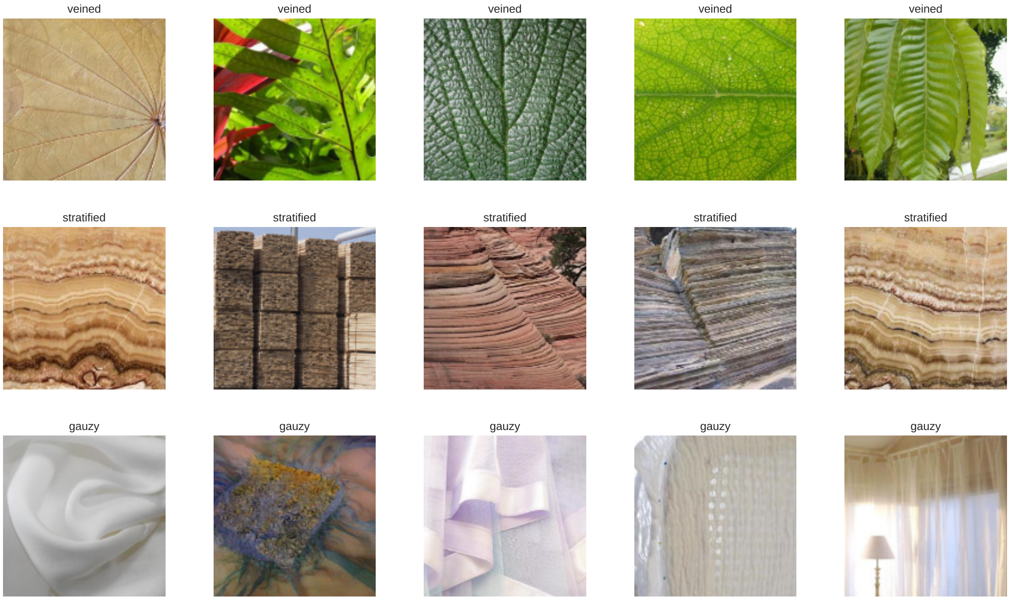


Figure 5.1: A sample of images from the DTD. One category per row is displayed. It's evident the great intra-class variability.

5.2 Classification Results

The procedure described in Chapter 4 was applied to the database. Images were all scaled and cropped to a square size $S = 256\,\text{px}$. Scattering network features were extracted for different combinations of J and L. The one-vs-rest linear SVM classifier was trained in an 8-fold cross-validation. Parameters that give higher mean accuracy in the cross-validation were searched.

Table 5.1 reports achieved accuracies. Overall, classification performance are poorer than those reported in [4]. However, this result was supposable because scattering convolutional networks offer an unspecialized representation of the input signal. Moreover, it's possible that the linear classificator that follows the representation is not enough to assimilate the great abstracted variance included in this database. Finally, it's worth noting that SVM are, by construction, a binary classifier. In this case, the generalization used for the multi-class problem is the one-vs-rest. Further studies could try other methods to achieve a better discrimination of the multiple classes, like the one-vs-one approach or the Crammer-Singer's method [36].

To conclude this simple first test, the combination of hyper-parameters that gave the best accuracy was used to train the model on the whole train set. Accuracy was then measured on the test set that was picked at the beginning of the procedure. The overall result is reported in Table 5.2. Accuracy grew a little, probably because more training samples are used in this last train.

S	J	L	C	Accuracy
256	6	6	$1 \cdot 10^{-6}$	0.194
256	6	6	$1 \cdot 10^{-4}$	0.299
256	6	6	$1 \cdot 10^{-2}$	0.256
256	6	8	$1 \cdot 10^{-6}$	0.214
256	6	8	$1 \cdot 10^{-4}$	**0.316**
256	7	6	$1 \cdot 10^{-6}$	0.155
256	7	6	$1 \cdot 10^{-4}$	0.272
256	7	6	$1 \cdot 10^{-2}$	0.263
256	7	8	$1 \cdot 10^{-6}$	0.165
256	7	8	$1 \cdot 10^{-4}$	0.294
256	7	8	$1 \cdot 10^{-2}$	0.262
256	8	6	$1 \cdot 10^{-6}$	0.127
256	8	6	$1 \cdot 10^{-4}$	0.238
256	8	6	$1 \cdot 10^{-2}$	0.265
256	8	8	$1 \cdot 10^{-6}$	0.138
256	8	8	$1 \cdot 10^{-4}$	0.256
256	8	8	$1 \cdot 10^{-2}$	0.277

Table 5.1: Performance evaluated with 8-fold cross-validation. Best accuracy is in bold. S: input image size. J: n.scales of scattering networks filters. L: n.rotations of scattering networks filters. C: cost parameter of SVM.

S	J	L	C	Accuracy
256	6	8	$1 \cdot 10^{-4}$	**0.329**

Table 5.2: Performance of the final model. S: input image size. J: n.scales of scattering networks filters. L: n.rotations of scattering networks filters. C: cost parameter of SVM.

CHAPTER 6

Melanoma Classification

6.1 Introduction

Malignant melanoma is a type of cancer that originates almost always from melanocytes. These are pigment cells of the skin, responsible for the production on melanin and thus bound to the skin color.

Several type of malignant melanoma (referred to herein only as “melanoma”) can be characterized basing on many features, for example the diameter, the irregularity of the borders and the spreading. Most common are: superficial spreading, lentigo malignant, nodular, acral lentiginous, mucosal, desmoplastic.

The melanoma is a commonly curable cancer if detected in early stage [37]. The primary treatment is excision biopsy, that can be performed to completely remove the lesion. Further surgery will assessment the presence of eventual near metastasis. In this case, the treatments can achieve a 98 % 5-year survival rate. If an accurate diagnosis is not conducted early enough, the disease can reach the lymphatic system and thus spread through the body. In this circumstance, the survival rate drops to 62%. The worst case is the creation of metastases around the body: the survival rate falls to 16%.

First diagnosis is commonly performed with a non-invasive visual inspection. Expert dermatologist performs the exam using the dermoscopy, a technique that employs a dermatoscope [38]. This tool is a light-controlled, high-resolution magnifying imaging device that can show clearly the lesion to the dermatologist. Dermatoscope works at direct contact to the skin using a layer of gel applied to the skin or without any contact.

The use of a transparent medium layer between the instrument and the skin permits to have a better illumination of the subject lesion.

To recognize a melanoma among other skin lesions, experts usually follow classification schemes. One of them is the “ABCDE rule”, where “A” is for asymmetry, “B” for borders, “C” for color, “D” for diameter, “E” for evolving. Odd behaviours of these features relative to a lesion can warn about the possible presence of a melanoma. More recently, other methods have obtained importance as simpler and/or more accurate substitutes of the “ABCDE rule”. For example, studies shown that “pattern analysis” outperform other approaches [39]. This method consists in searching predefined patterns on the lesion that can be described by common analogous (e.g. “honeycomb”) and that are more often associated with a melanoma. Overall, an expert dermatologist with the help of a dermatoscope can achieve an absolute accuracy between 75 % and 84 % [37].

Nowadays, the wide-spread adoption of dermatoscopes allows to have a constantly growing number of labelled high-resolution digital images that can be used to train machine learning algorithms. CAD (Computer Aided Diagnosis) systems, based on these algorithms, could help dermatologists to perform better and faster melanoma classification.

Many approaches followed in the years. A summary could be found in [40] and [37]. First implementations were based on the automation of classical schemes, by extracting hand-coded visual features like colors, size, shape and jagged borders. Features were than used to train a common classifier, directly or after the encoding with techniques like BoVW. More recent works were instead based on deep learning approaches, supported by the larger availability of labelled images. Some methods also used the interesting approach of transfer learning, where visual features learned on datasets of real-world images are used to fine-tune deep models on the specific task of melanoma classification.

Unfortunately, there is still a great discrepancy between results of different papers on the subject. Many metrics can be found in literature, ranging from the mere accuracy score, to more complex metrics like AUC or F-measures. Moreover, validity and reproducibility of results are often arguable because of the small and/or not representative size of the employed datasets.

An attempt to standardize the sector of automatic melanoma classification comes from the “ISIC Archive” (explained in the next section). One classification result based on this dataset can be found in [37]. In this work, authors reached a 93.1 % of accuracy (with 94.9 % sensitivity and 92.8 % of specificity) in melanoma vs. non-melanoma classification, using a mixed transfer-learning method. Other recent results comes from a challenge proposed at ISBI 2016 (International Symposium on Biomedical Imaging). The “Skin Lesion Analysis Towards Melanoma Detection” challenge saw 78 submissions from 38 groups [41] that contended in several phases. The challenge was based on a subset of the complete archive, composed of 900 training images and 379 test images. Among all the submissions, the best lesion classification result obtained was of 85.5 % of accuracy (with 50.7 % sensitivity and 94.1 % of specificity).

6.2 Database

A database obtained from the "International Skin Imaging Collaboration: Melanoma Project" (ISIC) was used to train and evaluate the classification performance of a model based on scattering networks.

The ISIC is a project of the "International Society for Digital Imaging of the Skin", with the purpose of publishing an open source public archive of skin images, which could help to improve diagnostic skills of physicians, to educate common people about melanoma and, above all, to create a big database for developers to train and test their algorithms for skin cancer detection.

ISIC-Archive is publicly available online [42]. The archive is organized in datasets that are collections of different studies from several clinics around the world. At time of writing, the archive hosts 12086 color images of skin lesions, of which 11301 are benign, 770 malignant cancer and 15 of unknown type. Of these, 9251 images are benign melanocytic moles in children. Lesions diagnosis is guaranteed on either biopsy-confirmation, a documented history of being clinically benign over several moths or independent reviews of four expert dermoscopists. Table 6.1 shows how each dataset is composed.

Dataset	Total Images	Benign	Malignant	Unknown
ISIC_UDA-1	557	401	156	-
ISIC_UDA-2	60	23	37	-
ISIC_MSK-1	683	445	223	15
ISIC_MSK-2	1534	1180	354	-
ISIC_SONIC-1	9251	9251	-	-

Table 6.1: Summary of images available at time of writing on the ISIC-Archive

Beyond malignant or benign classification, other clinical information are bundled with each image, like the age of the patient, sex and localization of the lesion.

Along with clinical metadata, the ISIC-Archive contains segmentation data of each image. These data were not used in this work, so that the proposed model could be evaluated with the original image without any additional information.

Lesion images and metadata were downloaded from the ISIC-Archive using the API interface provided by the website [1]. A Python script was created to automate this task.

The dataset containing moles of children was not used in successive analysis because it did not contain any malignant sample and so it could introduce a bias in the model. Images with unknown diagnosis were also dropped.

Overall, the database prepared for the analysis included 2049 benign (negative) examples and 770 malignant (positive) examples, for a total of 2819 images. Same sample

[1] `https://isic-archive.com/api/v1`. Accessed on 11/18/2016.

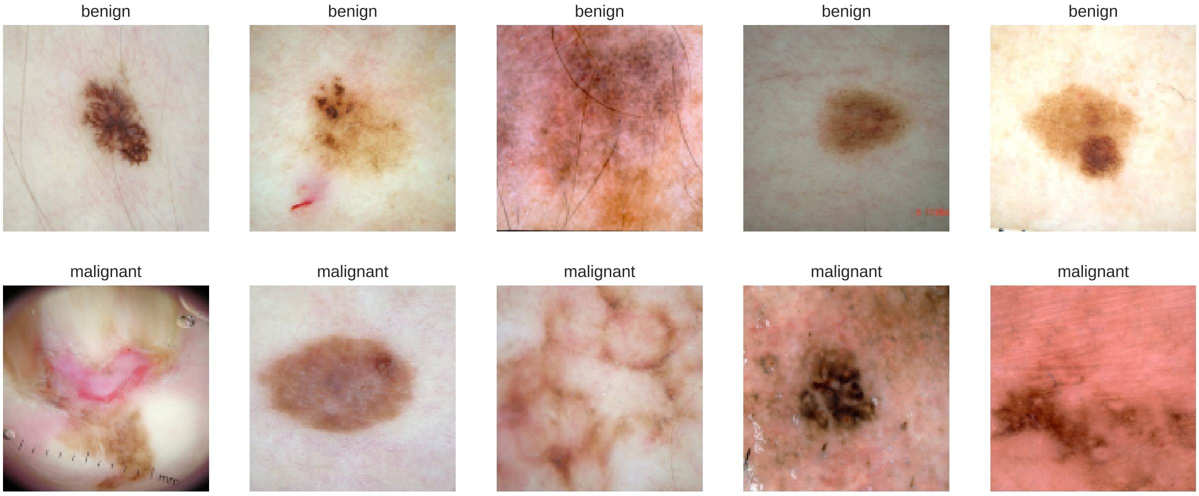


Figure 6.1: Some examples of images found in the ISIC Archive. Images in top row are benign lesions and those in the bottom row are malignant melanoma. Some frequent artifacts can be seen. In the top center image, the lesion is partially occluded by some hairs. In bottom left image, ticks of a ruler cover the lesion and some black borders are present. In fourth image of bottom row, some gel bubbles create light glares.

images from the database are shown in Figure 6.1.

The two classes turned out to be highly imbalanced (as seen in Table 6.2). This difference must be considered during training and evaluation of the classifier.

Class	N. Samples	Percentage (%)
Benign (negative)	2049	72.7
Malignant (positive)	770	27.3
Total	2819	100.0

Table 6.2: Summary of images available at time of writing on the ISIC-Archive

One issue found while preparing the database for the subsequent training was that sample images were not all the same shape. Figure 6.2 shows histogram of pixels count expressed in Megapixels. More than 1500 samples are 1024 px × 768 px = 0.75 Mpx images from the MSK-2 dataset. The MSK-1 dataset forms the peak at about 27 Mpx, while other datasets are spread at sizes lower than 10 Mpx. The preprocessing phase (exposed in Section 4.2) took care of this problem.

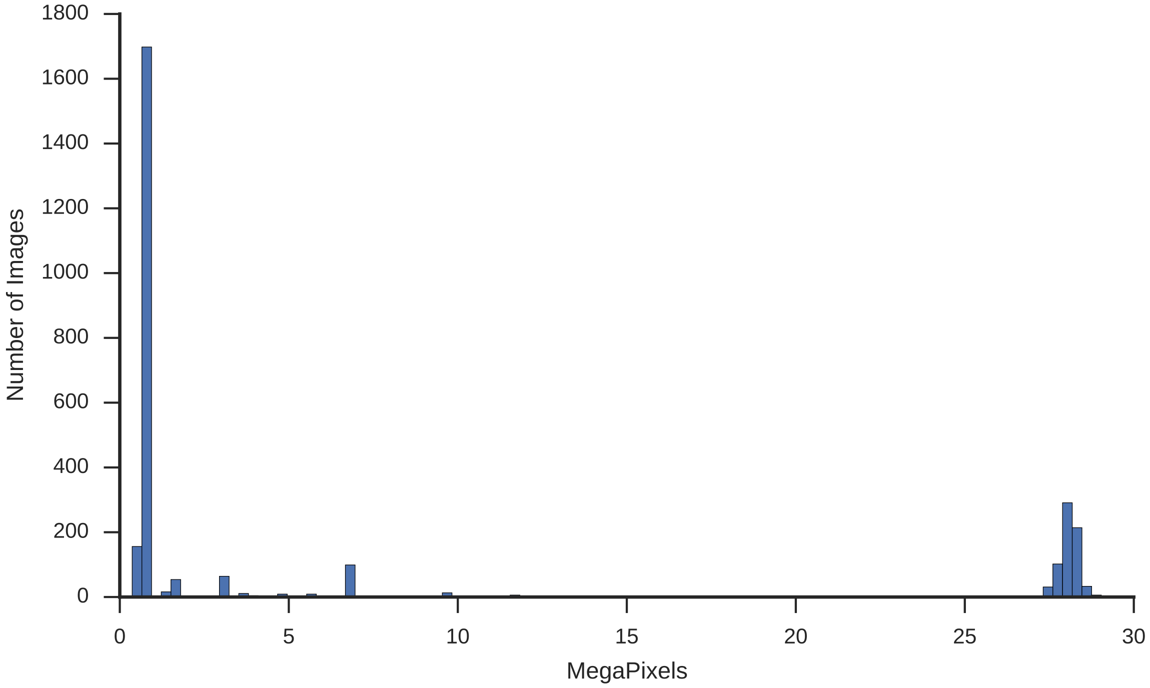


Figure 6.2: Histogram of image resolution of samples used for the training.

6.3 Model training and results

The selected subset of the ISIC database was used to train a model composed of a scattering network representation and a linear SVM, as exposed in Chapter 4.

It is worth noting that, as opposed to many authors, images were not prepared in order to remove artifacts like hairs, gel bubbles or other artificial signs. Moreover, the segmentation of the images is not used to confine the region of the lesion. This approach is followed in order to consider the scattering network a general purpose approach, possibly suitable for a wide range of problems without particular customizations.

To obtain a unique size for the images, they were resized to squares of side $S = 256\,\mathrm{px}$ and $S = 512\,\mathrm{px}$. The cross-validation method was used in order to find best values for: size S, filter scales J, filter orientations L and SVM cost C.

Table 6.3 reports accuracy, sensitivity, specificity, precision and ROC AUC for many combinations of the hyper-parameters, estimated with an 8-fold cross-validation. First observation is that despite the great range of parameters, there is not a great variability between metrics. Secondly, it can be seen that metrics are generally below state-of-the-art results reported in previous works.

Starting from the last point, results are quite those expected: the trained model is a general purpose system, not engineered in any way to consider the fact that it is trying to classify melanomas. Therefore poor metrics compared to ad-hoc system are ordinary.

Going into details of the first consideration, the accuracy on the train folds of the cross-validation was studied too. This value reached quite always high values, next to the 100 %, especially for larger C. With the linear classifier used, this means that the problem could be separated nearly linearly. This may be due to the "curse of dimensionality", a phenomena that appears when having a high number of features and a (relative) low number of examples. In this case the classification problem appears to be linearly separable. SVM have the theoretical background to guarantee that the hyper-plane with larger margin is found to try to reduce this "curse", but an optimal trade-off between misclassification cost and hyper-plane margin must be chosen. This is achieved by optimally selecting the parameter C. Subsection 4.6.1 explained the behaviour of the decision function on varying C.

In fact, it can be observed that values of C inside the range 10^{-4} and 10^{-2} gives a bit higher performance than other values. In this case the model appears to generalize better on the various validation folds. Therefore, a fine tuning on C was performed for best combinations of other hyper-parameter found in the previous grid-search, that is $S = 512$, $J = 7$ and $L = 8$. Results are shown in Table 6.4. A minor improvement on accuracy, precision and ROC AUC score has been achieved, passing the limit of 70 % of accuracy.

It is worth noting that other combinations of parameter J and L did not scored much worse results than the best configuration found. This is an interesting property of the scattering network representation that can be observed also on the texture dataset. Usually, the best configurations of the scattering network are for at least 6 rotations (hyper-parameter L) and a scale J such that $S/k \approx 2^J$, where $k \in \{1, 2, 3\}$. This behaviour has been seen also in other works that used this representation. This can be due to the fact that size of the scattering coefficients is approximately $S/2^J$, depending on the original size and S on the scale of the filters J. Therefore, if that size is small, the maximum translation invariancy is included in the representation, and better discriminability is obtainable. This usually makes the research of hyper-parameters very limited to few couples of J and L, speeding up the best model selection procedure.

Despite that, some more tests could be performed with different settings of the classifier in order to obtain better results. For example, an uneven weight could be applied to the misclassification cost of each class to try to adjust the trade-off between sensitivity and specificity. In fact, in the medical field, it's usually better to have a higher sensitivity (i.e. recognizing all positive/malignant samples). Nevertheless, it can be observed that the precision is always below 50 %, meaning that most of the lesions considered malignant are instead benign.

A different approach could use a classifier that includes non-linearities, for example a SVM with Radial Basis Function Kernel or a multi-layer perceptron. Non-linearities could allow to learn more complex relations between the features. Despite that, this

S	J	L	C	Accuracy	Sensitivity	Specificity	Precision	ROC AUC
256	7	6	$1 \cdot 10^{-6}$	0.641	0.625	0.647	0.410	0.636
256	7	6	$1 \cdot 10^{-4}$	0.682	0.740	0.660	0.460	0.700
256	7	6	$1 \cdot 10^{-2}$	0.683	0.708	0.674	0.459	0.691
256	7	6	$1 \cdot 10^{0}$	0.668	0.643	0.678	0.439	0.661
256	7	6	$1 \cdot 10^{1}$	0.653	0.606	0.671	0.419	0.638
256	7	8	$1 \cdot 10^{-6}$	0.640	0.659	0.632	0.413	0.646
256	7	8	$1 \cdot 10^{-4}$	0.681	0.731	0.661	0.458	0.696
256	7	8	$1 \cdot 10^{-2}$	0.688	0.672	0.695	0.463	0.683
256	7	8	$1 \cdot 10^{0}$	0.658	0.613	0.675	0.425	0.644
256	7	8	$1 \cdot 10^{1}$	0.644	0.599	0.662	0.409	0.630
256	8	8	$1 \cdot 10^{-6}$	0.614	0.566	0.633	0.377	0.600
256	8	8	$1 \cdot 10^{-4}$	0.669	0.711	0.653	0.445	0.682
256	8	8	$1 \cdot 10^{-2}$	0.683	0.695	0.679	0.459	0.687
256	8	8	$1 \cdot 10^{0}$	0.659	0.616	0.675	0.426	0.646
256	8	8	$1 \cdot 10^{1}$	0.654	0.618	0.668	0.422	0.643
512	7	6	$1 \cdot 10^{-6}$	0.658	0.659	0.657	0.429	0.658
512	7	6	$1 \cdot 10^{-4}$	0.692	0.724	0.679	0.469	0.702
512	7	6	$1 \cdot 10^{-2}$	0.693	0.665	0.704	0.468	0.684
512	7	8	$1 \cdot 10^{-6}$	0.665	0.672	0.662	0.438	0.667
512	7	8	$1 \cdot 10^{-4}$	**0.695**	0.738	0.679	**0.474**	**0.708**
512	7	8	$1 \cdot 10^{-2}$	0.691	0.651	**0.707**	0.465	0.679
512	8	6	$1 \cdot 10^{-6}$	0.648	0.638	0.651	0.417	0.645
512	8	6	$1 \cdot 10^{-4}$	0.683	0.738	0.662	0.461	0.700
512	8	6	$1 \cdot 10^{-2}$	0.678	0.688	0.674	0.453	0.681
512	8	6	$1 \cdot 10^{0}$	0.670	0.627	0.687	0.440	0.657
512	8	8	$1 \cdot 10^{-6}$	0.642	0.652	0.638	0.414	0.645
512	8	8	$1 \cdot 10^{-4}$	0.682	**0.742**	0.659	0.460	0.700
512	8	8	$1 \cdot 10^{-2}$	0.685	0.656	0.697	0.459	0.676
512	8	8	$1 \cdot 10^{0}$	0.667	0.597	0.695	0.434	0.646

Table 6.3: Performance evaluated with 8-fold cross-validation. Best metric are in bold. S: input image size. J: n.scales of scattering network filters. L: n.rotations of scattering network filters. C: cost parameter of SVM. ROC AUC: Receiver Operating Characteristic Area Under Curve

S	J	L	C	Accuracy	Sensitivity	Specificity	Precision	ROC AUC
512	7	8	$1 \cdot 10^{-6}$	0.665	0.672	0.662	0.438	0.667
512	7	8	$5 \cdot 10^{-6}$	0.673	0.704	0.661	0.449	0.683
512	7	8	$1 \cdot 10^{-5}$	0.681	0.722	0.665	0.457	0.693
512	7	8	$5 \cdot 10^{-5}$	0.689	0.735	0.671	0.466	0.703
512	7	8	$1 \cdot 10^{-4}$	0.695	**0.738**	0.679	0.474	0.708
512	7	8	$5 \cdot 10^{-4}$	**0.703**	0.722	0.695	**0.481**	**0.709**
512	7	8	$1 \cdot 10^{-3}$	0.696	0.699	0.695	0.473	0.697
512	7	8	$5 \cdot 10^{-3}$	0.691	0.668	0.700	0.466	0.684
512	7	8	$1 \cdot 10^{-2}$	0.691	0.651	**0.707**	0.465	0.679

Table 6.4: Fine-tuning over C. Performance evaluated with 8-fold cross-validation. Best metric are in bold. S: input image size. J: n.scales of scattering network filters. L: n.rotations of scattering network filters. C: cost parameter of SVM. ROC AUC: Receiver Operating Characteristic Area Under Curve

approach was not followed in order to keep the model usable for a generic class of tasks. In fact, the introduction of non-linearities usually implies that others hyper-parameters should be searched. The dependence of classification results with respect to these hyper-parameters is usually very unstable, therefore more robust, extended and task specific tunings should be performed.

With regard to the SVM with RBF kernel, this approach was tried nevertheless to confirm what was aforementioned in Subsection 4.6.1. Scattering network representation creates a high dimensional space, where number of features is usually larger than the number of samples. When this happens, classification performance of a linear kernel are usually comparable to those of a non-linear one. This was tested by training a SVM with RBF kernel with different combinations of parameters. Performance achieved were comparable to those obtained with the linear SVM, therefore the latter model was chosen in order to keep make the model more general purpose.

Finally, the parameters that allowed to achieve the best results of accuracy and ROC AUC in cross-validation were used to setup the final model. This model was trained with all the 70 % of the whole database that was put aside before for the training phase. The trained model was finally tested on the test set and performance metrics are reported in Table 6.5.

Final performance of the model are a little better than those found with the cross-validation. Presumably the larger size of the train set allowed the model to assimilate better the features space.

An ending note is about computational performance. The selected model is computational heavy. The feature extraction step required the usage of the *Titan X* GPU (see Section 3.6) because of its large amount of memory (more than 4 GB were necessary).

S	J	L	C	Accuracy	Sensitivity	Specificity	Precision	ROC AUC
512	7	8	$1 \cdot 10^{-6}$	**0.705**	**0.722**	**0.700**	**0.443**	**0.711**

Table 6.5: Performance of the final model. S: input image size. J: n.scales of scattering network filters. L: n.rotations of scattering network filters. C: cost parameter of SVM. ROC AUC: Receiver Operating Characteristic Area Under Curve

The scattering network in this configuration outputs 22416 features for a grayscale image, therefore 67248 for a RGB image.

Nevertheless, the total duration of the inference of the model is about 1 s per image with the proposed GPU implementation. This time permits to have a near real-time classification of a mole. An automatic model of this kind could be used in an application where a fast and gross classification is needed. Nevertheless, a better sensitivity is mandatory in the medical field, therefore some more studies for a better classification should be performed before.

Finally, a simpler configuration of the parameter could be used to train a model capable of running on a consumer-level GPU, like the one of a laptop. For example, the model with $S = 256, J = 7, L = 6, C = 1 \cdot 10^{-4}$ in Table 6.3 achieved performance comparable to the selected one, while being much simpler (with 9590 features produced by the scattering network).

CHAPTER 7

Conclusions

In this work, the scattering convolutional network has been studied in the context of machine learning. A scattering convolutional network permits to build a signal representation that is invariant to translations and stable to deformations.

After summarizing the theoretical structure and predominant qualities of this transformation, a high efficiency GPU code implementation has been proposed. The code takes advantage of the parallel architecture of GPUs to optimize most of the operations involved in the algorithm, especially the crucial convolutions with filters. This was implemented in the form of the convolutional theorem, by leveraging on the highly parallelizable Fast Fourier Transform and matrix multiplication. The implementation has been successfully tested for compatibility with the serial code published by the group of the authors of the scattering network representation. Moreover, high performance has been achieved, by reaching a speedup up to 20× with respect to the original code. The reduction of the computing time would allow nearly real-time application of this representation and more efficient future studies.

The scattering network has been applied to the classification task in two problems. Firstly, a test was conducted on a recent texture dataset to evaluate representation capabilities of the scattering network on a noticeably abstract classification problem. A linear SVM has been used as the classifier of the model. Results were inferior to the best models evaluated on the dataset, but considering the complexity of the database, the general purpose model based on scattering network obtained interesting outcomes.

Finally, a practical medical problem have been challenged: the malignant melanoma recognition. Melanoma is one of the deadest skin tumor, but if it is identified early the chance of recovering are substantial. A novel dataset of skin lesions has been used to train the same previous model based on scattering network representation and linear SVM. The model produced interesting classification results despite not being a system specialized in the task.

Overall the scattering network has proved to be an interesting method of transforming generic raw images in order to create a representation useful for subsequent analysis, particularly in classification tasks.

Hopefully the proposed GPU code will allow researchers to perform more studies on this representation and to discover more fields of application.

Bibliography

[1] Renato Campanini, Danilo Dongiovanni, Emiro Iampieri, Nico Lanconelli, Matteo Masotti, Giuseppe Palermo, Alessandro Riccardi, and Matteo Roffilli. A novel featureless approach to mass detection in digital mammograms based on support vector machines. *Physics in Medicine and Biology*, 49(6):961, 2004.

[2] Stéphane Mallat. Group invariant scattering. *Communications on Pure and Applied Mathematics*, 65(10):1331–1398, 2012.

[3] Joan Bruna and S. Mallat. Invariant scattering convolution networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1872–1886, aug 2013.

[4] M. Cimpoi, S. Maji, I. Kokkinos, S. Mohamed, , and A. Vedaldi. Describing textures in the wild. In *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[5] Isdis — international society for digital imaging of the skin. `http://isdis.net/`. (Accessed on 11/18/2016).

[6] Bioretics - beyond experience. `http://www.bioretics.com/`. (Accessed on 11/18/2016).

[7] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[8] Thomas Hofmann, Bernhard Schölkopf, and Alexander J Smola. Kernel methods in machine learning. *The annals of statistics*, pages 1171–1220, 2008.

[9] Stephane G Mallat. A theory for multiresolution signal decomposition: the wavelet representation. *IEEE transactions on pattern analysis and machine intelligence*, 11(7):674–693, 1989.

[10] Edouard Oyallon and Stéphane Mallat. Deep roto-translation scattering for object classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2865–2873, 2015.

[11] Xiuyuan Cheng, Xu Chen, and Stéphane Mallat. Deep haar scattering networks. *Information and Inference*, 5(2):105–133, apr 2016.

[12] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154, 1962.

[13] Mauro Ursino and Giuseppe Emiliano La Cara. A model of contextual interactions and contour detection in primary visual cortex. *Neural Networks*, 17(5-6):719–735, jun 2004.

[14] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[15] Karas Pavel and Svoboda Davi. Algorithms for efficient computation of convolution. In *Design and Architectures for Digital Signal Processing*. InTech, jan 2013.

[16] Gilbert Strang and T Nguyen. *Wavelets and filter banks*. Wellesley, MA : Wellesley-Cambridge Press, rev. ed edition, 1997. Includes bibliographical references (p. [475]-486) and index.

[17] John L Semmlow and Benjamin Griffel. *Biosignal and medical image processing*. CRC press, 2014.

[18] Cuda — nvidia. `http://nvidia.com/cuda/`. (Accessed on 11/18/2016).

[19] Programming guide :: Cuda toolkit documentation. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`. (Accessed on 11/16/2016).

[20] Best practices guide :: Cuda toolkit documentation. `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#axzz4QBiX3ke4`. (Accessed on 11/16/2016).

[21] Stéfan van der Walt, S Chris Colbert, and Gaël Varoquaux. The NumPy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, mar 2011.

[22] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012.

[23] Lev E. Givon, Thomas Unterthiner, N. Benjamin Erichson, David Wei Chiang, Eric Larson, Luke Pfister, Sander Dieleman, Gregory R. Lee, Stefan van der Walt, Teodor Mihai Moldovan, Frédéric Bastien, Xing Shi, Jan Schlüter, Brian Thomas, Chris Capdevila, Alex Rubinsteyn, Michael M. Forbes, Jacob Frelinger, Tim Klein, Bruce Merry, Lars Pastewka, Steve Taylor, Feng Wang, and Yiyin Zhou. scikit-cuda 0.5.1: a Python interface to GPU-powered libraries, December 2015. `http://dx.doi.org/10.5281/zenodo.40565`.

[24] G. Bradski. *Dr. Dobb's Journal of Software Tools.*

[25] IEEE standard for floating-point arithmetic.

[26] Michael L. Overton. *Numerical computing with IEEE floating point arithmetic.* Society for Industrial and Applied Mathematics, 2001.

[27] Comparing floating point numbers, 2012 edition — random ascii. `https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/`. (Accessed on 11/15/2016).

[28] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[29] Kristin P Bennett and Olvi L Mangasarian. Robust linear programming discrimination of two linearly inseparable sets. *Optimization methods and software*, 1(1):23–34, 1992.

[30] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[32] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[33] Mu-Chu Lee, Wei-Lin Chiang, and Chih-Jen Lin. Fast matrix-vector multiplications for large-scale logistic regression on shared-memory systems. In *Data Mining (ICDM), 2015 IEEE International Conference on*, pages 835–840. IEEE, 2015.

[34] Wei-Lin Chiang, Mu-Chu Lee, and Chih-Jen Lin. Parallel dual coordinate descent method for large-scale linear classification in multi-core environments. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 1485–1494, New York, NY, USA, 2016. ACM.

[35] Kristin J Dana, Bram Van Ginneken, Shree K Nayar, and Jan J Koenderink. Reflectance and texture of real-world surfaces. *ACM Transactions on Graphics (TOG)*, 18(1):1–34, 1999.

[36] Koby Crammer and Yoram Singer. On the algorithmic implementation of multiclass kernel-based vector machines. *Journal of machine learning research*, 2(Dec):265–292, 2001.

[37] Noel Codella, Junjie Cai, Mani Abedini, Rahil Garnavi, Alan Halpern, and John R. Smith. Deep learning, sparse coding, and SVM for melanoma recognition in dermoscopy images. In *Machine Learning in Medical Imaging*, pages 118–126. Springer Science Business Media, 2015.

[38] M. Binder, M. Schwarz, A. Winkler, A. Steiner, A. Kaider, K. Wolff, and H. Pehamberger. Epiluminescence microscopy. A useful tool for the diagnosis of pigmented skin lesions for formally trained dermatologists. *Arch Dermatol*, 131(3):286–291, Mar 1995.

[39] P. Carli, E. Quercioli, S. Sestini, M. Stante, L. Ricci, G. Brunasso, and V. De Giorgi. Pattern analysis, not simplified algorithms, is the most reliable method for teaching dermoscopy for melanoma diagnosis to residents in dermatology. *Br. J. Dermatol.*, 148(5):981–984, May 2003.

[40] Michel Fornaciali, Micael Carvalho, Flávia Vasques Bittencourt, Sandra Avila, and Eduardo Valle. Towards automated melanoma screening: Proper computer vision & reliable results, 2016.

[41] David Gutman, Noel C. F. Codella, Emre Celebi, Brian Helba, Michael Marchetti, Nabin Mishra, and Allan Halpern. Skin lesion analysis toward melanoma detection: A challenge at the international symposium on biomedical imaging (isbi) 2016, hosted by the international skin imaging collaboration (isic), 2016.

[42] Isic archive. `https://isic-archive.com/`. (Accessed on 11/18/2016).