SPATIAL PYRAMID KERNEL FOR IMAGE CLASSIFICATION

Machine Learning II

David Bergés, Roser Cantenys and Alex Carrillo

November 2019

Universitat Politècnica de Catalunya

Contents

Chapter 1

Introduction

Abstract

This paper presents a method for recognizing toy pieces categories based on the global features that describe color and illumination properties, and by using the statistical learning paradigm. This technique works by partitioning the image into increasingly finer sub-regions and computing a weighted histogram intersection in this space. First, we show the mathematical properties of it to be used as a kernel function for Support Vector Machines (SVMs). Second, we describe the implementation and give examples of how these SVMs, equipped with such a kernel, can achieve very promising results on image classification.

1.1 Aim and motivations

The contribution of this paper is twofold. On the one hand, the principal aim of this project is to study a non-standard kernel, in our case it will be in image domain, and we apply it to a specific problem of our interest, with a kernel method. The focus in this task is on the kernel function and on its application. In addition, through it, we put into practise and consolidate the knowledge learned during the first part of the course and the information acquired when consulting papers about new kernels. Furthermore, thanks to the theoretical aspects studied in this course, we are able to understand distinct kernels, prove its associated properties and, consequently, confirm that they are kernels themselves. In this manner, this project will help us to have a better interpretation of kernel methods.

On the other hand, the main motivation of this project is the fact of working with images. We have concluded this since we have never worked with them before and we are excited about testing pictures. Over and above, we think that the results will be more interpretable and intuitive. In addition, we think that it is an interesting problem considering that nowadays there is plenty of images in the net and we would like to categorise or classify them. We implement this kernel from scratch and using Python programming language. After crawling the web and studying different papers, we decided to study *Pyramid match kernel* and apply it in a support vector machine in distinct datasets.

Chapter 2

Theoretical aspects

2.1 Kernel learning approaches for image classification

This report examines the use of a kernel learning technique to a specific problem of image classification. In image classification one aims to separate images based on their visual content. Kernel learning is a paradigm in the field of machine learning that generalizes the use of inner products to compute the similarity between objects.

The contributions we report are specific for this task and the application to a more general class of problems is not a trivial issue. In the following we will discuss the main ideas behind spatial kernels, in particular the one we have choose to address this problem and then show how our paradigm can be applied to this class of kernels.

2.2 Spatial pyramid matching

The *spatial pyramid matching* framework can be seen as a schematic representation of image information. More specifically, a spatial pyramid is a collection of orderless feature histograms computed over cells defined by a multi-level recursive image decomposition. In other words, this named pyramids aim to incorporate spatial information of the image.

Possibly the simplest way to represent color information (also considering a gray scale) is provided by histograms. That is because different images give rise to different representations that enhance or attenuate certain color features. In spite of their simplicity, histograms are very efficient for many practical tasks, as they are stable to changes of view. Nevertheless, it must be taken into account that all information regarding spatial features and spatial correlation is discarded. This leads to invariance to rotations and tolerance to translations, which can become an issue for certain sets of images. This property will be discussed and tested throughout the report.

Taking this approach of spatial pyramids into account, at level 0 the decomposition of an image consist of just a single cell, and the representation is equivalent to a standard bag of features, that is, using the (frequency of) occurrence of each element (*i.e.* a pixel) of the picture as a feature for training a classifier. At level 1, the image is subdivided into four quadrants, yielding four feature histograms, and so on. Spatial pyramids can be matched using the *pyramid kernel*, which weights features at deeper levels more highly, reflecting the fact that higher levels localize the features more precisely.

2.2.1 Pyramid match kernel

The spatial pyramid scheme is implemented as follows. We subdivide the image into L different levels of a pyramid, enumerated by $\ell = 0, 1, \ldots, L-1$. The level ℓ corresponds to a $2^\ell \times 2^\ell$ grid on the image plane, for a total of $D = 2^{d\ell}$ non overlapping cells of equal size, where d is the dimension of the data, in these case $d = 2$. In this construction higher levels of the pyramid correspond to a finer griding of the picture. Then, we build a histogram for each cell in a level ℓ singly which results in D histograms.

Let H_X^ℓ and H_Y^ℓ denote the histograms of X and Y at level ℓ, so that $H_X^\ell(i)$ and $H_Y^\ell(i)$ are the number of points from X and Y that fall into the ith cell of the grid. Then the number of matches at level ℓ is given by the *histogram intersection* function:

$$\mathcal{I}(H_X^\ell, H_Y^\ell) = \sum_{i=1}^{D} min\left(H_X^\ell(i), H_Y^\ell(i)\right) \tag{2.1}$$

We have to remind that the intersection is defined as the minimum of two histograms. In addition, to simplify, in the following, we will abbreviate $\mathcal{I}(H_X^\ell, H_Y^\ell)$ to $\mathcal{I}^\ell$. Figure 2.1 illustrates the above.

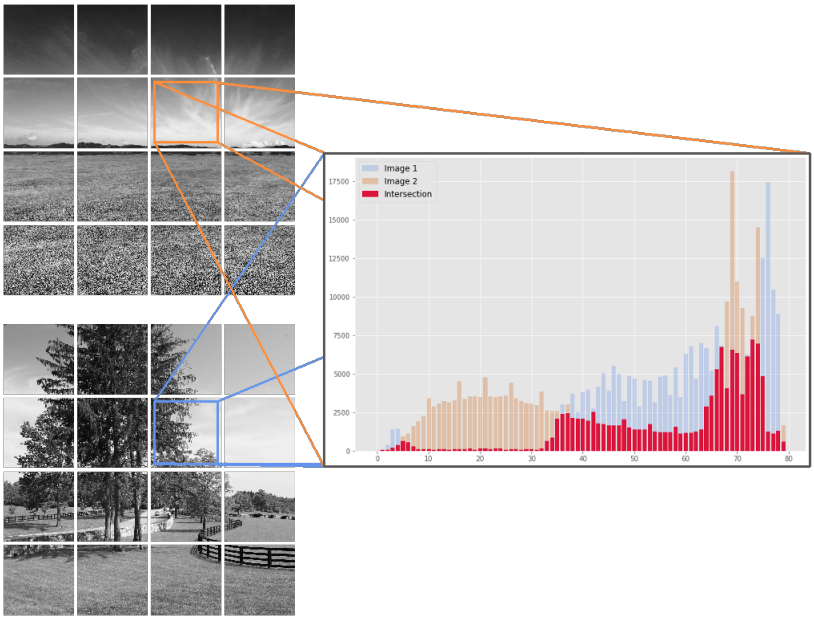

Figure 2.1: Intersection of the histograms in cell $(2, 3)$ of two images with $\ell = 4$. Both cells are defined by distinct histograms, the orange one and the blue one, respectively. We observe that, in fact, the intersection of the two corresponds to the minimum value of each bin: the red area.

Note that the number of matches found at level ℓ also includes all the matches found at the finer level $\ell + 1$. Therefore, the number of new matches found at level ℓ is given by the difference between two consecutive levels, that is $\mathcal{I}^\ell - \mathcal{I}^{\ell+1}$ for $\ell = 0, \ldots, L-1$. However, the weight associated is different for each level. It is inversely proportional to cell width at that level. It means that we penalize matches found in larger cells because they involve increasingly dissimilar features and reward matches found in smaller cells. So the weight associated to each level is set to $\frac{1}{2^{L-\ell}}$.

Putting all the pieces together, we get the following definition of the *pyramid match kernel*:

$$\begin{aligned}
\kappa^L(X, Y) &= \mathcal{I}^L + \sum_{\ell=0}^{L-1} \frac{1}{2^{L-\ell}} (\mathcal{I}^\ell - \mathcal{I}^{\ell+1}) \\
&= \mathcal{I}^L + \frac{1}{2^L}(\mathcal{I}^0 - \mathcal{I}^1) + \frac{1}{2^{L-1}}(\mathcal{I}^1 - \mathcal{I}^2) + \frac{1}{2^{L-2}}(\mathcal{I}^2 - \mathcal{I}^3) + \cdots + \frac{1}{2^{L-(L-1)}}(\mathcal{I}^{L-1} - \mathcal{I}^L) \\
&= \frac{1}{2^L}\mathcal{I}^0 + \left(\frac{1}{2^{L-1}} - \frac{1}{2^L}\right)\mathcal{I}^1 + \left(\frac{1}{2^{L-2}} - \frac{1}{2^{L-1}}\right)\mathcal{I}^2 + \cdots + \left(\frac{1}{2^{L-L}} - \frac{1}{2^{L-(L-1)}}\right)\mathcal{I}^L \\
&= \frac{1}{2^L}\mathcal{I}^0 + \frac{1}{2^{L-1+1}}\mathcal{I}^1 + \frac{1}{2^{L-2+1}}\mathcal{I}^2 + \cdots + \frac{1}{2^{L-L+1}}\mathcal{I}^L \\
&= \frac{1}{2^L}\mathcal{I}^0 + \sum_{\ell=1}^{L} \frac{1}{2^{L-\ell+1}}\mathcal{I}^\ell
\end{aligned} \tag{2.2}$$

Both the histogram intersection and the pyramid match kernel are Mercer kernels.

As stated above, a pyramid match kernel works with an orderless image representation, which means that the generated histogram is independent of the position of the pixels of the original picture (discards all spatial information). This technique allows to represent an image with a collection of features in a high-dimensional space as large as the range of histogram values. However, it can be interesting to quantize all feature vectors into M discrete types, and make the simplifying assumption that, given the representation of two images, only features of the same type can be match to one another.

The parameter M can be seen as the number of bins used in the histogram and, this way, each channel m gives us the pyramid match kernel of the m bin. The final kernel is then the sum of the separate channel kernels:

$$K^L(X,Y) = \sum_{m=1}^{M} \kappa^L(X_m, Y_m) \tag{2.3}$$

This approach has the advantage of maintaining continuity. In fact, it reduces to a standard bag of features when L = 0.

2.2.2 Satisfying Mercer's condition

Mercer's theorem is a representation of a symmetric positive-definite function on a square as a sum of a convergent sequence of product functions. This theorem is used to characterize a symmetric positive semi-definite kernel and exposes what conditions determine which functions can be considered as kernels. It states that an optimal solution to a kernel-based algorithm is guaranteed to satisfy that

(i) the kernel function must be symmetric

(ii) the kernel matrix K (also called Gram matrix) must be positive semi-definite, *i.e.* PSD.

If both conditions are satisfied then the matrix element K_{ij}, which corresponds to the $k(i,j)$ function, can be a kernel function. Moreover, the Gram matrix merges all the information necessary for the learning algorithm: the data points and the mapping function are fused into the inner product.

The *symmetry condition* (i) is proved in a forward step:

$$\begin{aligned}
K^L(X,Y) &= \sum_{m=1}^{M} \kappa^L(X_m, Y_m) = \sum_{m=1}^{M} \left(\frac{1}{2^L} \mathcal{I}^0 + \sum_{\ell=1}^{L} \frac{1}{2^{L-\ell+1}} \mathcal{I}^\ell \right) \\
&= \sum_{m=1}^{M} \left(\frac{1}{2^L} \left(\sum_{i=1}^{D} min\left(H_X^0(i), H_Y^0(i)\right) \right) + \sum_{\ell=1}^{L} \frac{1}{2^{L-\ell+1}} \left(\sum_{i=1}^{D} min\left(H_X^\ell(i), H_Y^\ell(i)\right) \right) \right) \\
&= \sum_{m=1}^{M} \left(\frac{1}{2^L} \left(\sum_{i=1}^{D} min\left(H_Y^0(i), H_X^0(i)\right) \right) + \sum_{\ell=1}^{L} \frac{1}{2^{L-\ell+1}} \left(\sum_{i=1}^{D} min\left(H_Y^\ell(i), H_X^\ell(i)\right) \right) \right) \\
&= \sum_{m=1}^{M} \kappa^L(Y_m, X_m) = K^L(Y,X) \qquad \square
\end{aligned} \tag{2.4}$$

The *positive semi-defined condition* (ii) requires to expound on several ideas.

As the kernel function is a sum of histogram's intersections, we just need to prove that a histogram intersection is a kernel inasmuch as the sum of kernels is a kernel as we have proved in class. We proceed to prove it by showing that histogram intersection is an inner product in a suitable feature.

Let H_X^ℓ and H_Y^ℓ be the histograms of images X and Y. Now we will represent H_X^ℓ and H_Y^ℓ with an $(N \times 1)$-dimensional binary vector, $\mathbf{H}_\mathbf{x}^\ell, \mathbf{H}_\mathbf{y}^\ell$, respectively. Each vector is composed of $M \times N$ positions where M is the number of bins of the histogram and N is the maximum height of the histogram. Every N positions we will have a new bin. Each bin will be represented by h_m ones and $N - h_m$ zeros, for $m \in 1, ..., M$, where the ones represent each count of a given bin as we can see below.

$$\mathbf{H}_\mathbf{X}^\ell = (\overbrace{1\ 1 \cdots 1}^{h_{x_1}}, \underbrace{0 \cdots 0}_{N-h_{x_1}}, \cdots, \overbrace{1\ 1 \cdots 1}^{h_{x_m}}, \underbrace{0 \cdots 0}_{N-h_{x_m}}, \cdots, \overbrace{1\ 1 \cdots 1}^{h_{x_M}}, \underbrace{0 \cdots 0}_{N-h_{x_M}})$$

$$\mathbf{H}^{\ell}_{\mathbf{Y}} = (\overbrace{1\,1\cdots1}^{h_{y_1}}, \underbrace{0\cdots0}_{N-h_{y_1}}, \cdots, \overbrace{1\,1\cdots1}^{h_{y_m}}, \underbrace{0\cdots0}_{N-h_{y_m}}, \cdots, \overbrace{1\,1\cdots1}^{h_{y_M}}, \underbrace{0\cdots0}_{N-h_{y_M}})$$

As we expressed both histograms as vectors, now the histogram intersection can be seen to be equal to the inner product between the two corresponding vectors $\mathbf{H}^{\ell}_{\mathbf{x}}, \mathbf{H}^{\ell}_{\mathbf{y}}$.

$$\mathcal{I}^{\ell}(H^{\ell}_x, H^{\ell}_y) = \mathbf{H}^{\ell}_{\mathbf{x}} \cdot \mathbf{H}^{\ell}_{\mathbf{x}} \tag{2.5}$$

We apply the previous result to show that the intersection is PSD. To prove it we use that every component of vectors $\mathbf{H}^{\ell}_{\mathbf{x}}, \mathbf{H}^{\ell}_{\mathbf{y}}$ are ≥ 0, since $\mathbf{H}^{\ell}_{\mathbf{x,i}}, \mathbf{H}^{\ell}_{\mathbf{y}}, \in \{0, 1\}$.

$$c^{\top}\mathcal{I}c = c^{\top} \cdot {\mathbf{H}^{\ell}_{\mathbf{X_i}}}^{\top} \cdot \mathbf{H}^{\ell}_{\mathbf{Y_j}} \cdot c \geq 0 \qquad \square \tag{2.6}$$

As the histogram intersection is a kernel since is symmetric and positive semi-defined, so is the kernel we proposed because it is a sum of the histogram intersection kernel.

2.3 Computational complexity

In computer science, the complexity of an algorithm is the amount of resources required for running it. As the amount of needed resources varies with the input, the complexity is generally expressed as a function of $n \to f(n)$, where n is the size of the input, and $f(n)$ represents the worst-case scenario, *i.e.* the maximum of the amount of resources that are needed for all inputs of size n. When the nature of the resources is not explicitly given, this is usually the time needed for running the algorithm, *i.e. time complexity*. On the other hand, the other case that is usually taken into account is the *space complexity*, which stands for the amount of memory that will be used for running our algorithm.

We will explicitly talk about the implementation of the kernel in the following chapters, but how all these complexity studies fit into our specific problem? Without going too much in detail, we will use a `Python` library called `sklearn`. This library allows us to define our own kernels by either giving the kernel as a Python function or by precomputing the Gram or kernel matrix. Both options involve implicitly computing the latter matrix and therefore we will study the complexity of this last operation.

In this paper we will refer by complexity to how long it takes to calculate the kernel matrix. Being n the size of the input, the kernel matrix will be $n \times n$, and thus the amount of elements to compute grows quadratically. Asymptotically speaking the complexity of computing it will be $O(n^2)$, but as in almost every practical use-case of the kernel n will not tend to infinity, consequently we can use the fact that the matrix is symmetric, *i.e.* $K_{ij} = K_{ji}$ halving the total amount of computations. Nevertheless, this complexity is irreducible since we cannot reduce the number of items to compute. For the purpose of reducing the computational time, we focus on reducing the execution time of each iteration, *i.e.* the time needed to compute each element of the kernel matrix, K_{ij}. As our kernel is not a simple dot product, where its iterations take an insignificant amount of time, it contrarily takes few milliseconds depending on the CPU unit that is being used.

The calculation of each element depends on the size of the image, the amount of binning applied and the partition level L. Now we study the influence of these parameters on the computation time. As far as parameter L is concerned, the time complexity grows as the number of partitions of the image with an extra cost, an overhead, made by the partitioning of the image. In addition, as we can see, time computation is conditioned by the size of the image. If we have a low resolution, the complexity time required to compute the histogram will be lower. We also need to be aware of the size and partition of the image, we would like to have images of size power of two to be able to partition it without any troubles. Furthermore, if the number of bins increases, the intersection time also increases.

Chapter 3

Dataset

3.1 Set of images

The dataset used in this project is *Images of Lego Bricks* which we have taken from Kaggle. This dataset is composed by 6379 grayscale images which are divided in 16 *LEGO*® brick classes shown in Figure 3.1. Each brick was selected in Mecabricks.com and next imported in collada (.dae) format in Blender. They used an animator object to render the imported brick which contains 400 images taken from 400 different angles. There is an exception, *2357 Brick corner 1x2x2*, which only has 379 images. All pictures are in a gray scale so we just have one channel instead of three, *i.e.* the RGB channels. This in fact, will ease our implementation as we will not be dealing with the three separate histograms for each image, and it will eventually speed up our running times by a factor of 3. The size of each image is 200×200, thus low-medium-resolution.

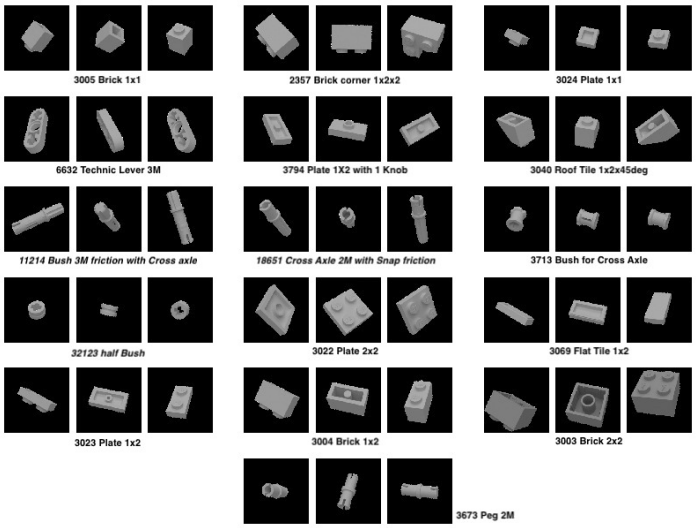

Figure 3.1: Samples of three angles of view for all the 16 classes of pieces in the dataset.

The dataset contains two identical folders so we just discard one and keep the other one that will be divided into training and validation sets according to the requirements. Both folders contain 16 identified subfolders with the images of a given class, which greatly simplifies the labeling process .

3.2 Data preprocessing

With this data, we are not prepared to input these images into any model directly but we need to process them and generate a suitable representation. The idea is to build a dataframe by flattening the pixel values of an image in a vector shape of size 40.000 (result of the picture dimension 200×200px), adding the target variable which determines the class of the image (the name of the piece) and organizing each sample in rows.

It should be noted that even though all the images are in `.png` format, meaning that they have an alpha channel that represents the degree of opacity of a color (in particular, their backgrounds are transparent), they have been converted to a black background when importing them. In this manner, the range of gray values of the numerical variables, *i.e.* the pixels, is limited to the interval $[0, 255]$, which extremities signify black and white, respectively.

Chapter 4

Implementation and difficulties

Python is an interpreted, high-level, general-purpose programming language. Its design philosophy emphasizes readability, and it supports multiple programming paradigms, including procedural, object-oriented, and functional programming. Benefits that make Python our choice for this work include simplicity and consistency, access to great libraries and frameworks for AI and machine learning, flexibility, platform independence and a wide community. Moreover, taking into account our great experience with the language all of these made Python the best fit for our project.

Further down we will debate some of the key aspects of the implementation, clarify all the decisions taken and explain all the difficulties encountered.

4.1 Data loading and partitioning

As mentioned in section 2.3, one of the factors that affects the performance of our model the most is the size of the images. In this paper we carefully selected a dataset that had a reduced square 200x200 resolution, thus enabling every image to be represented as a flattened vector in $\mathbb{R}^{40000}$. Thinking ahead, plotting the different images in different downgraded resolutions (64×64 and 128×128, as shown in Figure 4.1) using inter nearest-neighbour interpolation, we noticed that the loss in resolution did in fact not affect as much the interpretation of the image, and it allowed us to reduce the whole dimensionality of the problem making each histogram intersection computation much faster.

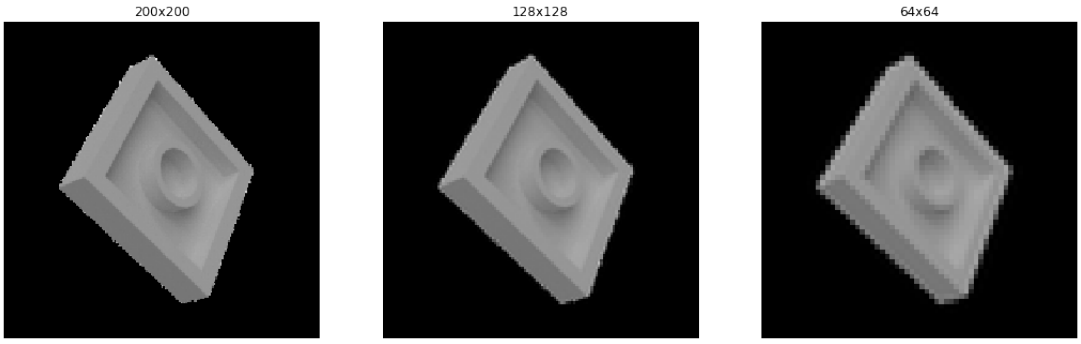

Figure 4.1: Sample of piece *3022 Plate 2x2* of the dataset from a fixed angle, represented in three different resolutions, 200×200, 128×128 and 64×64, respectively. It can clearly be seen that a 136 times reduction of the data causes a perceptible blur to the lines. Nevertheless, the visual difference is not huge but memory saving is widely achieved.

We will specifically discuss the histogram computation problem in one of the following sections, but the underlying idea is that computing a histogram from an array of dimension 40000 is way more expensive than, for example, doing it with one of reduced dimension 4096.

It should be noted the intelligent partitioning of the images, since they will be square with a resolution that is a power of 2, avoiding special cases of odd partitions and thus making the code implementation simpler. If we had to deal with odd partitioning, we would have had to implement a more flexible code enabling different block sizes, or by simply adding some kind of padding to the image. Hereafter, there is a representation example of the latter shown in Figure 4.2.

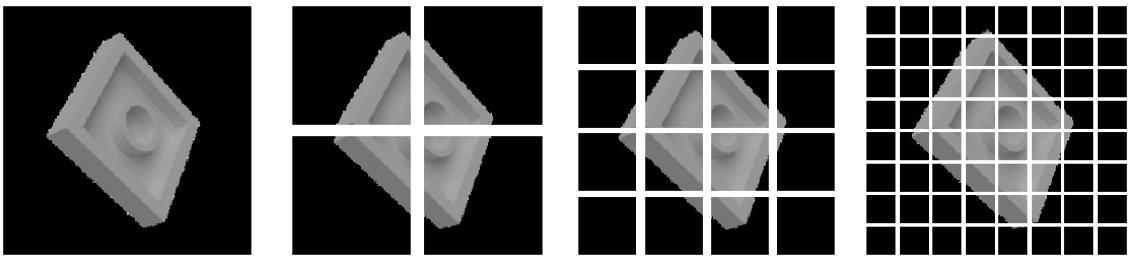

Figure 4.2: Partitioning levels of a sample image from the dataset, with depth $L = 3$.

4.2 Binning

Hence that by converting the .png images to a gray scale, we reduced all the potential values for each pixel in between a range from 0 to 255. That could be interpreted as if the histogram intersection would at most be of dimension 255, in the worst case scenario. Taking that into account, if we reduced the total amount of bins from each histogram, *i.e.* aggregating different pixel colors into a smaller range, we would potentially reduce the dimensionality of the intersection, and thus making computations faster. In fact, we are quantizing our images, we are mapping input values from a large set (range from 0 to 255) to output values in a countable smaller set. Let us illustrate the latter with an example: if the total amount of desired bins is 10, we would map the whole 255 pixel values making each bin have around 25 pixel tonalities.

Of course, one cannot forget about the potential loss of accuracy that this transformation could convey. In practice with this specific dataset, as we will see in the next chapter Results, the loss of accuracy will not be as big as one could expect. Notice that images are rather monotonic, with a big black background and similar shades of gray in the center (reprsenting the lego figures), and therefore their histograms will have a huge peak at value 0 and some others in between the spectrum. This restricts the range of gray scale pixels, thus making the dimension of the histogram of at most 80. This reduced range of pixel tonalities means that the reduction of precision due to binning will not be as sizeable as in other case scenarios, but in fact, plently speeding up our executions.

4.3 Kernel

In order to implement the kernel function itself, we adopted the same definition of the final *pyramid match kernel*, the 2.3 formulated expression. This function will be used when building the kernel matrix, as $\mathbf{K_{ij}} = K^L(i, j)$, which we will describe in the following subsections. The above is the same as saying that each entry of the matrix will call that function. For that reason, reducing runtime when creating the histograms, computing their intersection and building the kernel matrix is crucial.

4.3.1 Histograms computation

As the creation of a histogram for each cell is a recurrent function used at every level ℓ of partitioning up to L, the appropriate structure to compute and storage them is very meaningful. To address this very real problem, and after several trial an test implementations, we made use of the `Counter` Python function, which is a tool provided to support convenient and rapid tallies. Broadly speaking, with it, elements are stored as dictionary keys and their counts are stored as dictionary values. However, it allows common patterns for working with this class of objects, such as converting them into to type `set` and provides several mathematical operations including the intersection of their corresponding counts.

With this approach in mind, the implementation of the kernel function is straightforward. The sum of the weighted intersections is simple and the optimization in time is remarkable. In addition to this, in the next section we will discuss the importance of precalculating all $L + 1$ histograms for a given image sample in order to avoid redundant computations.

4.3.2 Building the matrix

Implementing kernel SVM with *Scikit-Learn* is similar to the simple SVM. This method is very versatile as different kernel functions can be specified for the decision function and common kernels are provided, but it is also possible to specify custom kernels. You can define your own kernels by either giving the kernel as a Python function (callable) or by precomputing the Gram matrix. If a callable is given, it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape `(n_samples, n_samples)`.

In our case, we decided to implement the SVM by using the Gram matrix. At the moment, the kernel values between all training vectors and the test vectors must be provided. It should be noted that the computation of the matrix is not trivial; two scenarios must be considered. On the one hand, there is the case when we are training the SVM, *i.e.* X is equal to Y (with $[X|Y]$ representing the dataframe), where the Gram matrix is symmetric $G_{ij} = G_{ji}$ and thus we can halve the total number of computations, as mentioned in section 2.3. On the other hand, otherwise there is the case when the matrix is not symmetric, hence we are not in training. In such circumstances we can not reuse computations.

Lastly and most importantly, there is a part which requires special consideration regarding the optimization in the calculations of the Gram matrix entries. As stated, each element of the matrix is the result of applying the kernel function 2.3 to all samples of the dataset. For this reason, the fact of having the histogram matrices (or the histograms themselves) computed beforehand causes an enormous decrease in the execution time when building the kernel matrix. That nuance gave us a turnaround at the moment of trying out the whole implementation by reducing the waiting time and allowing us to load more data samples into the model.

Chapter 5

Experimenting and results

5.1 Playing with the dataset

As we pointed out in the previous chapter, depending on the size of the training dataset and the parameters of our kernel, the training time of our model grows exponentially. Nonetheless, we wanted to test our kernel with a Support Vector Machine taking a wide spectrum of parameters, but performing all these experiments in our computers would be unfeasible, that is why we considered a cleverer option. By using the power of cloud computing and taking advantage of the tools Python offers for creating multiple subprocesses we were able to overcome this barrier, and thus enabling us to run several SVMs in parallel in a virtual machine, each one with different parameters. In this way, we could optimize a lot the time spent on experimentation.

Hereafter, we show several results obtained combining all parameters mentioned before. We were unable to perform any kind of cross-validation in our tests, as runtime would have been too long. Therefore, we decided to partition our data forming a train and a test sets, with 50% and 20% of the observations respectively. We decided to stick with the default $C = 1$ parameter for the SVMs and only tunning the parameters of our kernel due to computation time limitations.

We split the results in two parts: one for each image resolution tested, 64×64 and 128×128, recall we have not experimented with the original resolution due to the fact that it is computationally more expensive and as the results will show, there is no significant evidence of performance loss with the respective downgraded resolutions. We will show two plots composed by a table and a graphic. The graphic shows the accuracy of the test set in function of L, quantification and time while the table highlights the best accuracies obtained.

L	quantization	time (min)	accuracy
4	20	84.90	0.808
4	30	80.65	0.807
4	40	88.16	0.799
3	20	21.90	0.795

Table 5.1: Best results of 64×64

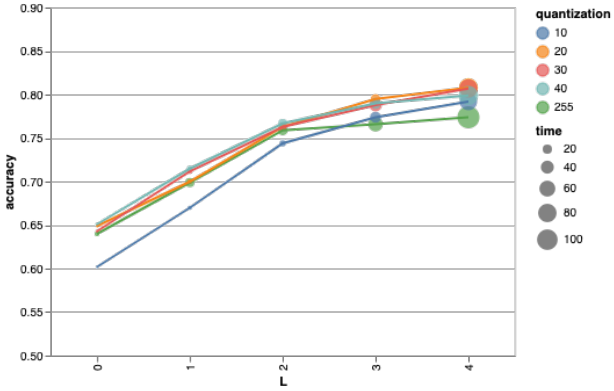

Figure 5.1: Images 64×64

L	quantization	time (min)	accuracy
3	30	33.58	0.803
3	40	39.70	0.802
4	30	75.92	0.806
4	40	78.26	0.806

Table 5.2: Best results of 128×128

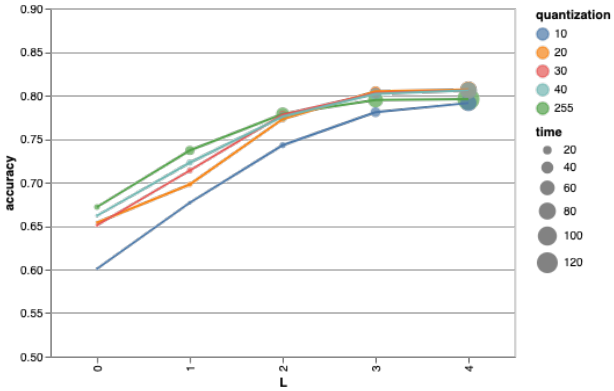

Figure 5.2: Images 128×128

As we can see, the L parameter influences a lot its accuracy. The higher the L the better the accuracy we get. Nevertheless, once we reach $L = 3$ we do not obtain relevant improvements. In addition, we can observe that this parameter is intrinsically related to the computation time. As we increase L, the time grows considerably. However, the time is not only determined by L, it is also influenced by the quantization. For a same level , the lower number of bins we have, that is the higher quantization we do, the less computing time required, as expected. In addition, the number of bins of the quantization have influence over the accuracy. As we could expect for a major number of bins we get a major accuracy. In spite of that, we can see that the difference between using 40 or 255 bins is not that big. We conclude that it is because our images are quite plain, i.e. we do not have 255 bins different from 0, just 70 or 80 differ from 0. Consequently, the reduction applied is not as big as it seems, we are just reducing from 70 or 80 to 40, not from 255. That is why the accuracy does not grow as much as we anticipated with quantization. If we tested it with another dataset with a large range of colors, we would expect to get higher accuracy as we increased the number of bins.

As predicted, in general, a better resolution gives rise a better accuracy. Regardless of the better results being achieved with the highest resolution images, in this case, the best one is obtained with 64×64 resolution, $L = 4$ and quantization $= 20$.

Chapter 6

Conclusions

The main purpose of this project was to study a non standard kernel to be able to work with images. Not only we have studied its theoretical aspects but we have been able to implement it and perform several experiments as well.

The fact of having had to implement our kernel, has lead us to a better and deeper understanding of the internal functioning of the machine learning libraries implementing the SVM, and how the different kernels can affect its precision, speed and accuracy. Thus, we have arrived to the conclusion that it is crucial to optimise the kernel function by any means, as the runtime can grow exponentially with the size of the training observations, and consequently making it unfeasible to try-out our models.

Focusing more on our specific kernel, the *Spatial Pyramid Kernel*, we wind up that, in general, the size of the image is a very important factor, and it also represents an upper bound on what we can compute. For example, take a $4K$ image, that is 3840×2160, i.e. 8294400 pixels, making it fairly impossible for our kernel to run if there are a substantial amount of observations. In addition, we have seen that quantization may be very useful for some applications, as its precision loss in many cases is not significant enough and it improves runtime performance a large amount. What is true is that working with cloud computing becomes a very useful tool when having to execute huge workloads that for our computers would be simply too much.

It is true that we could obtain better results with other machine learning algorithms that work better with images such as convolutional neural networks. In spite of this, taking into account the simplicity of the idea behind our kernel it has been very interesting to check its surprisingly good performance and accuracy.

6.1 Improvements and further development

After this project, it would be a great idea to apply and modify this kernel in order to be able to utilize it in any kind of images. The first objective is to handle color images, RGB or RGBA. It would be a great idea to deal, separately, with three histograms, one for each component. However, we should be conscious that we would loss the correlation between colors and maybe we could think about a better implementation. The second main objective is to manage with no squared images, with rectangular partitions. It would imply to deal with sizes that are not power of two hence cope with odd partitions.

A clever solution for coping with bigger images or bigger datasets, would be the use of SVM ensemblings. The underlying idea would be to split the bigger training set, and to train different SVMs in parallel with a different split for each one, and in the end combining all the partial results to obtain an approximate solution.

References

[1] Annalisa Barla, Francesca Odone, and Alessandro Verri. "Histogram intersection kernel for image classification. (English) [Conference Paper]". In: *ICIP International Conference* (2003). URL: https://www.researchgate.net/publication/4044656_Histogram_intersection_kernel_for_image_classification.

[2] Kristen Grauman and Trevor Darrell. "The Pyramid Match Kernel: Discriminative Classification with Sets of Image Features. (English) [Article]". In: *Massachusetts Institute of Technology* (2005). URL: https://www.cs.utexas.edu/~grauman/papers/grauman_darrell_iccv2005.pdf.

[3] Svetlana Lazebnik, Cordelia Schmid, and Jean Ponce. "Beyond Bags of Features: Spatial Pyramid Matching for Recognizing Natural Scene Categories. (English) [Article]". In: (2007). URL: https://inc.ucsd.edu/~marni/Igert/Lazebnik_06.pdf.

[4] Svetlana Lazebnik, Cordelia Schmid, and Jean Ponce. "Spatial Pyramid Matching. (English) [Article]". In: (2007). URL: https://www.cs.unc.edu/~lazebnik/publications/pyramid_chapter.pdf.

[5] Peter Vincent Gehler. "Kernel Learning Approaches for Image Classification. (English) [Dissertation]". In: *Naturwissenschaftlich-Technischen Fakultät I, Universität des Saarlandes, Saarbrücken* (2009), pp. 83–121. URL: https://pdfs.semanticscholar.org/37d6/cde8be756b70d22262f1acc3442a0c6aa7ea.pdf.

[6] From Wikipedia the free encyclopedia. *Bag-of-words model*. 2012. URL: https://en.wikipedia.org/wiki/Bag-of-words_model.

[7] Dongping Tian, Xiaofei Zhao, and Zhongzhi Shi. "Support Vector Machine with Mixture of Kernels for Image Classification. (English) [Article]". In: *Institute of Computing Technology, Beijing, 100190, China* (2012). URL: https://link.springer.com/content/pdf/10.1007%2F978-3-642-32891-6_11.pdf.

[8] Massimiliano Patacchiola. *The Simplest Classifier: Histogram Comparison*. 2016. URL: https://mpatacchiola.github.io/blog/2016/11/12/the-simplest-classifier-histogram-intersection.html.

[9] Scikit Learn Developers. *C-Support Vector Classification, sklearn.svm.SVC*. 2019. URL: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html.

[10] Python Software Foundation. *High-performance container datatypes, collections*. 2019. URL: https://docs.python.org/2/library/collections.html.