MASTER THESIS

Handling variable shaped & high resolution images for multi-class classification problem

Author:
Yiannis Sotiropoulos

Tutor:
Prof. Javier Béjar Alonso

Supervisor:
Dario Garcia Gasulla (PhD)

Co-Supervisor:
Ferran Parés Pont (PhD candidate)

A thesis submitted in fulfillment of the requirements for the degree of Master in Artificial Intelligence in the

High Performance Artificial Intelligence Group *of*
Barcelona Supercomputing Center

April 20, 2020

Abstract

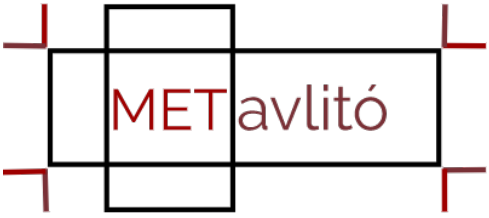
Facultat d'Informàtica de Barcelona
Barcelona Supercomputing Center

Master in Artificial Intelligence

Handling variable shaped & high resolution images for multi-class classification problem

by Yiannis Sotiropoulos

Convolutional Neural Networks (CNNs) are usually trained using a pre-determined fixed spatial image size. While scale-invariance is considered important for visual representations, CNNs are not scale invariant with respect to the spatial resolution of the input image; since a change in image dimension may lead to a non-linear change of their output. At the same time, there are applications (e.g. in medicine) where images come in multiple scales and shapes not leaving any space for applying common transformations with which images are deformed and shrinked losing important information. Leaving high-resolution information can be a big also burden, resource-wise, with high computational costs, memory and time requirements. Like that there has been a shift of focus in research from parameter optimization and connections readjustment towards an improved architectural design of the network; since different state of the art networks such as Xception, ResNext, PolyNet and others explore the effect of different transformations on CNNs' learning capacity. Instead of modifying the internals of CNNs *METavlitó* project focuses mainly on the pre-processing stage of the network in order to handle high-resolution images, as well as, the variability in their shape. *METavlitó* proposes two components, one for clustering images' resolution into buckets and a training component for scale invariant learning employing an input agnostic architecture decreasing the average GPU memory requirements. Compared to a classic approach which follows the common pre-processing transformations (resizing & cropping) before training, our solution, using the same architecture controls more the overfiting, increases the accuracy by $3-5\%$ and decreases the average GPU memory needs by approximately 43% and thus, the total duration of the training and validation time.



Keywords: Deep Learning, Expectation-Maximization, Clustering, KMeans, Agglomeratie, Classification, Computer Vision, Bucketing, Variable shaped, High Resolution, Padding, Deformation, Downratio, Dynamic resize, Dynamic crop

Acknowledgements

The current thesis project constitutes my last step of my master degree at Universitat Politecnica de Catalunya.

At first, I would like to thank my tutor Prof. Béjar and my supervisors Dario Garcia and Ferran Parés for giving me the opportunity to work on this project, the insightful suggestions and the whole colaboration. Finally, I would like to thank my family and friends who supported me throughout my studies.

Contents

List of Figures

List of Tables

List of Abbreviations

HR	**H**igh **R**esolution
DL	**D**eep **L**earning
ML	**M**achine **L**earning
CNN	**C**onvolutional **N**eural *N*etwork
TPA	**T**otal **P**adding **A**rea
RPR	**R**andom **P**adding **R**efinement
PC	**P**adding **C**ost
BS	**B**atch **S**ize
FC	**F**ully **C**onnected
OF	**O**bjective **F**unction
EM	**E**xpectation **M**aximization

Chapter 1

Introduction

1.1 Motivation

There is a rapid, revolutionary change in computer vision, caused by deep convolutional neural networks (CNNs) [24] and the availability of big datasets [9]. Deep learning approaches have recently been substantially improving upon the state of the art in image classification [23], [37], object detection [18], recognition [30] and other tasks. Existing deep CNNs require a fixed input image size which means that they do not take into account any variability of the shape (aspect ratio) neither of the scale of the input image. Typically, images are resized to $224x224$ since the appearance of CNNs with AlexNet[23]; according to the paper this size was also used to allow the extraction of random patches for translation invariance. At that time there were also hardware limitations that did not allow training with larger images. Researchers, though, continued using this size without any concrete theoretical basis only because empirically it was giving good results.

This requirement needs a previous transformation which can be obtained either via resizing and cropping [23], [37] or warping [18], [11]. Cropping may not contain the entire object, while the warped image results in geometric distortion. Both, often alter image composition, reduce image resolution, or cause image distortion (see *Figure 1.1*) resulting most likely in a decrease in accuracy for the images or sub-images of an arbitrary size/scale [27]. When images are deformed in cases like this, is not desirable for a set of domains which may require attention to detail like in autonomous cars, bio-medicine, etc.

Padding the image to a fixed shape, bigger than the biggest cropped image shape is another option according also to [27]. However, this is not investigated thoroughly and it is unknown if learning the extra padded area increase the performance of the classifier or not, but in general it is a way to solve deformation.

FIGURE 1.1: Common image transformation options

A dataset which includes variable shaped images and/or high resolution images like the ones in the *Figure 1.2*, can bring many different challenges in the performance of the CNN model not only from the perspective of accuracy and loss but also from the perspective of required resources (i.e. memory, GPU power) [1].

- Variable shape

  The variability in the shape of the image has to do with the image's aspect ratio (which is the fraction of width divided by the height of the image). Being able to take pictures from a wide variety of devices (like mobiles, tablets, etc) has created a bigger demand on handling images with different orientation/shapes. Most of the current approaches interpolate all images to ensure that they have the same shape. However, this approach leads to information loss and/or deformation [17]. Furthermore, there are some use cases where any form of loss of data can have multiple problems and drawbacks in some specific use cases. For example, in segmentation or localization for brain tumor classification, it is needed that proportions remain unchanged in medical images because it plays an important role the the place and the size a tumor is detected in [2].

- High resolution (HR)

  We consider high resolution images the ones which have large pixel dimension. In *METavlitó* we consider high resolution images which have at least one dimension with size larger than 1080 pixels. In the *Figure1.2* the left image has a dimension with lower resolution and one with extremely high resolution; the right image presents the same peculiarity, however, both dimensions are in high resolution.

  The vast majority of current image recording devices are capturing images minimum at 500x500 pixels. This resolution is almost double the size of the

common required size of the images mentioned above. There are many use cases based on HR images in multiple scales like in the medical domain for breast cancer [16], [26] or in robotics for autonomous cars [8], where resizing ends up in loss of information necessary for better predictions.

FIGURE 1.2: Horizontal (5632x944) and vertical (1920x6810) panorama images

1.1.1 Hypothesis Space

As we saw the value of handling variable-shaped images in HR in real-case scenarios, we build our project upon some hypothesis which was based on our research and it will be used to setup the framework we want to work on.

1. *Loss of information reduces CNN's performance*

   By resizing images we lose some details especially in more cluttered ones. Even in the example of the *Figure 1.3* the details can not be seen creating a confusion on what it depicts or the material is made of; in the example the small picture may look more like it is made of rock or even ivory instead of marble. We hypothesize that this can cause again decrease in the performance of the CNN in terms of accuracy.

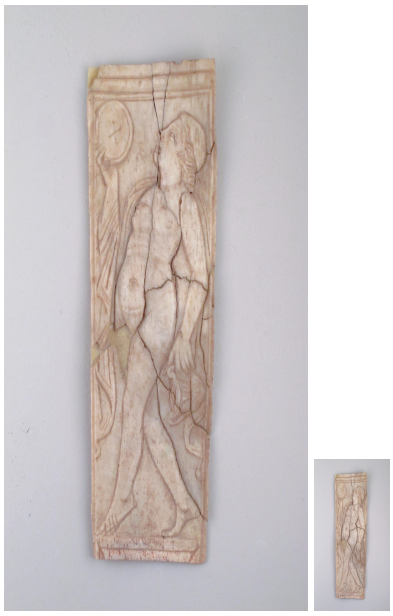


FIGURE 1.3: Results when losing information from an image

2. *Shape Deformation reduces CNN's performance*

   Deforming images like the ones in the example in *Figure 1.4* can affect the shape of the object which in turn can have some negative effects not only in object recognition but also in texture recognition. For this reason we hypothesize that deformation can cause a decrease in the accuracy.

FIGURE 1.4: Results of deformation

3. *Padding can increase CNN's performance with a slower convergence*

   Padding is mostly used in CNNs inside the layers so that frames near the edges of the image to contribute as much to the output as frames near the centre; but it is not used so much as a pre-processing technique. Here we will use it as a way to batch images together instead of just resizing to a pre-determined size. From the example in *Figure 1.5* we can see that here we actually add some information, the padded (black) area. Because of this we assume that the memory needs would probably increase, in case we batch an image with a larger one, like in the example. Because of cases like this we hypothesize that padding will

result in slower training convergence, but also in higher accuracy since we can leverage this to encapsulate more information (the shape of image) and we do not deform the images.

FIGURE 1.5: Desired padding example

4. *Excessive padding can decrease CNN's performance*

   In cases when large amounts of padding is applied in order to match the size of two images like in the example in *Figure 1.6* we expect to have much worse results because the biggest part of the padded image does not contribute to the network's learning. When this happens we hypothesize a decrease in the CNN's accuracy.

FIGURE 1.6: Extreme padding example

1.2 Problem Setup

1.2.1 Challenges

Working with variable-shaped images in HR entails several challenges that need to be addressed.

- *Input agnostic architecture*

CNNs mainly consist of convolutional layers and fully-connected layers that follow. The convolutional layers give feature maps of any size that represent the spatial arrangement of the activations as seen in *Figure 1.7*. That is because they operate in a sliding-window manner using kernel filters and that's why they do not require a fixed image size as input. The parameters to be found are the convolved weights/values of the filters, regardless of the input and thus of the amount of convolution. However, the fully-connected layers need to have fixed size of the input. Usually, the output of the last convolutional layer is flattened but then requires fixed input of the network. The techniques that usually are used to overcome this and make the architecture more agnostic of the input is through changing the dimensionality of the feature maps output of the last convolutional layer.

- Global Average Pooling, an operation that calculates just the average output of each feature map in the last convolutional layer
- Adaptive Average Pooling, is an operation where given the output size, it takes the input and automatically calculates the stride and kernel size to adapt the needs. This concept incetivized adaptive approaches to learn class-specific pooling shapes [33]. $Stride = (input\_size//output\_size)$ $Kernelsize = input\_size - (output\_size - 1) * stride$ $Padding = 0$

Both are fairly simple operations which reduce the data significantly and prepares the model for the final classification layer as it's used also in [29].

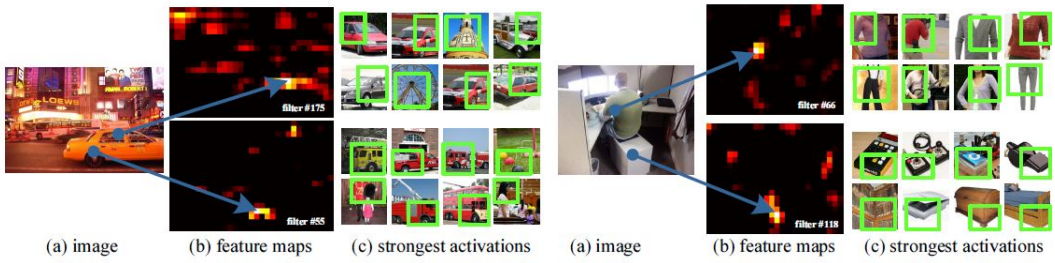


FIGURE 1.7: Visualization of the feature maps. (a) Two images in Pascal VOC 2007. (b) The feature maps of some convolutional filters. The arrows indicate the strongest responses and their corresponding positions in the images. (c) The ImageNet images that have the strongest responses of the corresponding filters. The green rectangles mark the receptive fields of the strongest responses. Source [13], [20]

- *Scale invariance*

  Scale invariance is needed in order to be able to handle images in multiple resolutions. It can be achieved by modifying the network structure [36] or artificially enriching the dataset through "data augmentation" [23] so as to increase the CNN's robustness to inputs of different scale. However, this gives more challenges because HR images still require a lot of memory and a lot of operations per second because of the size of feature maps which are the outputs of each convolutional layer.

- *Memory limitations*

  When the input images of a CNN are larger then the resulting feature maps of the convolutional layers are larger as well. This actually means that the

GPU that these images reside in needs to allocate more memory and also more power to calculate and save the activations for each image.

1.2.2 Related Work

The use of variable shaped images in high resolution contain a lot of information and requires a lot of memory especially if they are not resized. These aspects have been considered in the past, with some relevant contributions based on CNNs. One approach suggests to make input agnostic architectures for scale invariant learning through multiple scale evaluation during the post training like in the case of object detection [25] or segmentation [19]. In these tasks, a network that was pre-trained with fixed image size for classification is used as the backbone of a larger model that is expected to adapt to a wide variety of image sizes. Spatial pooling [20] proposes a new network structure for robust object deformations showing that it can improve the accuracy for some tasks; but, it doesn't work with high-resolution images and it doesn't mention the network's requirements in memory; as a fully CNN solution may need a lot of memory to train larger images.

Different modifications in CNNs are performed to make it appropriate for various image scales while trying to tackle resource limitations [21] too. There have been studies which try to deal with the conflicting demands of multi-scale and full-resolution dense prediction. The work of [26] involves repeated up-convolutions that aim to recover loss of information while carrying over the global perspective from down-sampled layers. Another approach involves providing multiple re-scaled versions of the image as input to the network and combining the predictions obtained for these multiple inputs [14]. However, it is not clear whether separate analysis of re-scaled input images is truly necessary.

In addition, prominent modifications of CNNs are knowledge distillation, training of small networks, or squeezing of pre-trained networks (such as pruning, quantization, hashing, Huffman coding, etc.) [34],[7],[15]. GoogleNet[31] exploited the idea of small networks, which replaces the conventional convolution with pointwise group convolution operation to make it computationally efficient. Similarly, ShuffleNet[12] used pointwise group convolution but with a new idea of channel shuffle that significantly reduces the number of operations without affecting the accuracy. In the same way, ANTNet[35] proposed a novel architectural block known as ANTBlock, which at a low computational cost, achieved good performance on benchmark datasets. No matter the advancements on that area though, there has not been any improvements tackling all of the challenges; most of the proposed solutions suggest some complex approaches addressing one or two of the challenges.

1.3 Research Objectives

The purpose of *METavlitó* project is to handle variable shape and high-resolution images in a classification problem with multiple classes using deep neural networks. The first step in that process is the pre-processing, which is about how data is fed into the neural network. This includes all the transformations on the raw data before it is fed to the deep learning algorithm. We will explore some dynamic transformations which can be employed in the pre-processing pipeline of the neural network and we evaluate them according to the performance of the classifier in terms of accuracy, time, required resources mainly average GPU utilization and GPU memory usage.

The main goal is to find a way to learn better hidden patterns/feature maps of images taking into account their initial shape and size. At the same time it is possible to reduce the average network's needs in resources, which can speed up training, reduce footprint and enable larger input sizes for recognizing more and richer details. Even if an architecture is able to process variable shapes of images, each batch still needs to be consistent for coherency in the computational operations; even having batches of 1 is not desirable because of computational inefficiency. For that reason, in this project we propose padding instead of fixed-size image resizing, as a way to batch together images of different shapes. Padding generates a space around the image in order to make it match the others in the same batch as in *Figure 1.8*. Furthermore, a technique was developed to minimize these padded areas in order to limit as much as possible this information which is added.

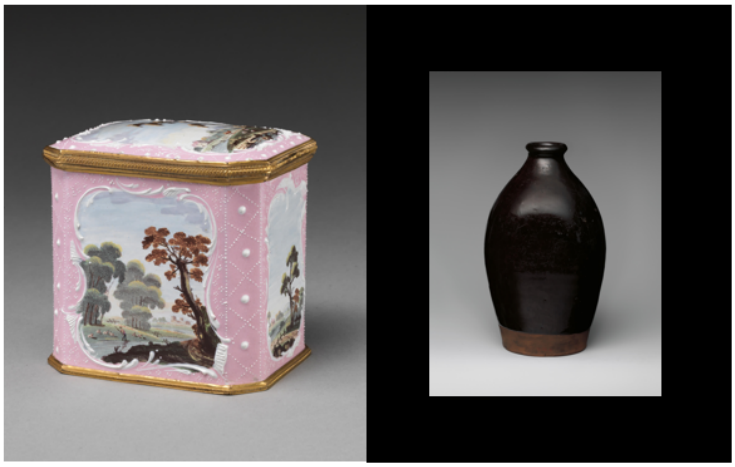


FIGURE 1.8: Example of how padding is used to batch together images of different sizes

Thus, the main topics which were explored towards that objective are the following:

- Dynamic transformations which take into account the aspect ratio (width/height) of each image without having to provide fixed-size dimensions.
- Minimize the padding area of the images in each batch.
- A technique so as to train together at the same forward and backward pass images with similar shapes and sizes.

1.4 Structure

The document is structured in a way to follow the reasoning and the decisions made during researching on this topic. At first, in Chapter2 we explain *METavlitó* project showing the concept behind its design and implementation. Later in Chapter3 we will show the dataset we used and the results of the experiments that we conducted in order to test some hypothesis that we made based on our research. After that, in Chapter4 we gather all the insights obtained and we make the conclusions and also we show some possible future extensions to improve the current approach. Finally, in Chapter5 is the Appendix with some theory which is provided for the reader for better understanding of the decisions made in Chapter2.

Chapter 2

Project: METavlitó

METavlitó comes from the abbreviation MET, the Metropolitan museum of Art of New York, and the greek word $\mu\varepsilon\tau\alpha\beta\lambda\eta\tau\acute{o}$ which means variable. The name was inspired from the origin and the type/characteristics of the datasets which were finally used in this research project and includes variable-shaped and high-resolution images.

2.1 Project design analysis

The design of a solution that can handle variable-shaped images in high resolution has some challenges. To design the implementation of the proposed approach we made certain hypothesis that we will try to validate later in the experiments. These hypothesis led us to some prerequisites of the implementation and on top of that we implemented some extra components as shown below in the Implementation Analysis.

METavlitó project wants to leverage on the image transformation techniques to increase the performance of the model. The research is focused more on the preprocessing stage which can be applied independently of the architecture used in the CNN; however, will effect the way the network learns by taking into consideration the shape and scale of the image.

According to Jeremy Howard, a data scientist and researcher in *fast.ai*, padding a big piece of the image (256x256 pixels) will have the following effect: the CNN will have to learn that the black part of the image is not relevant and does not help to distinguish between the classes (in a classification task), as there is no correlation between the pixels in the black part and belonging to a given class. As this is not hard coded, the CNN will have to learn it by gradient descent, and this might probably take some epochs.

Thus, the main concept of the implementation is to minimize the padding area in order to minimize the extra information and then based on that, to batch together images with similar shapes.

Given that, *METavlito* is composed of 2 components. The first thing we need to do is to minimize the padding area, which takes as input the dimensions of the

images and outputs the images arranged in a way that they fit in different separate, non-overlapping $\mathcal{K}$ buckets/groups. A bucket consists of a group of images with similar shapes, as it can be seen in the *Figure 2.4*.

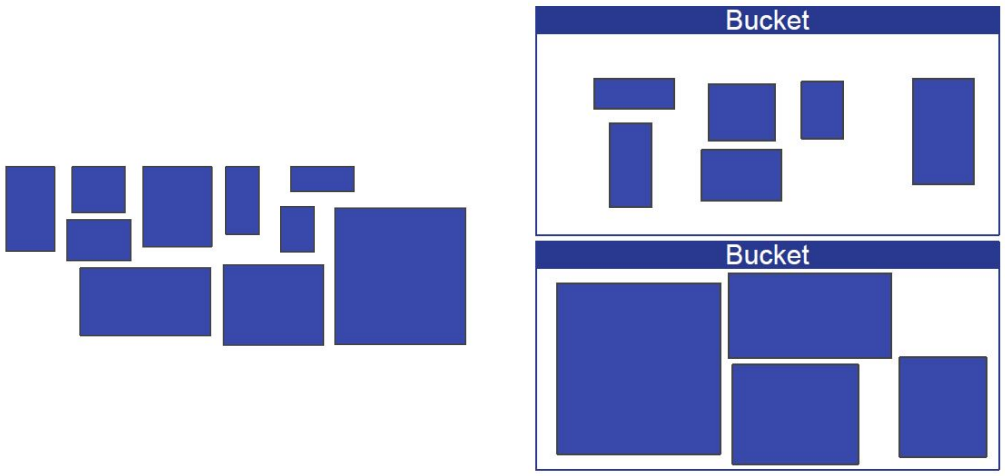


FIGURE 2.1: Turn datasets into buckets based on their shape & size

In these buckets we mind about the *total padding area* which is the total amount of pixels needed to pad each image of the bucket so as to have the same shape and size as seen in the *Figure 2.2*.

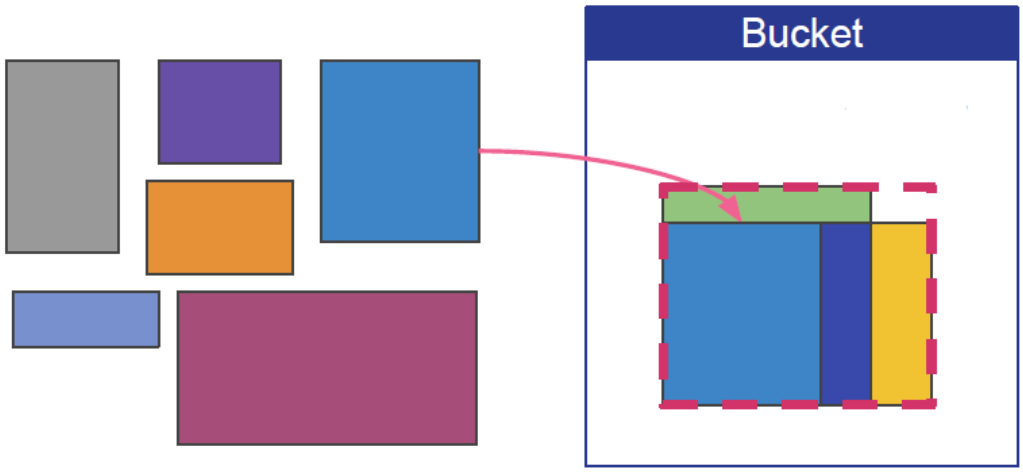


FIGURE 2.2: The way to pad images into buckets - Bucket Padding Area definition

As *Total Padding Area* (TPA) we define the following:

Given a set of images $\mathcal{I} = \{i_1, i_2, ..., i_n\}$ where $\forall\, i \,\epsilon\, \mathcal{I} : i(x, y), x, y \,\epsilon\, \mathbb{Z}_+$, we define a set of buckets $\mathcal{B} = \{b_1, b_2, ..., b_K\}$ where $\forall\, i \,\epsilon\, \mathcal{I}$: $i \,\epsilon\, b_j$, $j \,\epsilon\, 1, 2, ..., K$.

Note that $K \leq n$.

The *Padding Cost* (PC) of all the images of the set of buckets is:

$$TPA = \mathcal{PC}(\mathcal{B}) = \sum_{j=1}^{K} \mathcal{PC}(b_j) \tag{2.1}$$

Finally, we relate the PC of a bucket with its size, the number of elements (images) that it contains ($|b_j|$).

$$\mathcal{PC}(b_j) = |b_j| \cdot prod(max(b_j)[x,y]) - \sum_{\forall i \epsilon b_j} prod(i[x,y]), \tag{2.2}$$

where $prod(b_{max}[x,y]) = b_{max\_x} \cdot b_{max\_y}$

Ideally, buckets are so coherent that any other arrangement or even if a single image goes to another bucket will make the total padding area to rise. These requirements refer to an NP-Hard problem which led to clustering as the task which is aligned with these characteristics and could give an approximate solution for this part. Thus, in this part the clusters can be also called buckets and the clustering can be called bucketing and TPA is the replacement of the error metric (sum of the squared errors) which is used in the classic clustering.

The second thing we need to do is to batch together the images of the buckets during the training of the CNN. Typically, during training random batches of the dataset of specific size are created and trained through a forward and backward pass, $(BS, ImgX, ImgY, C)$ where BS is the batch size, $ImgX, ImgY$ is the image size and C is the number of channels. Also, there is some stochasticity in this procedure because typically, when training by (some variant of) mini-batch stochastic gradient descent, the elements of each batch should be sampled as uniformly as possible from the total input [6]; otherwise, it is possible that the network will overfit to whatever structure was in the input data, and the resulting network will not have a good performance. What we want to do in *METavlito* project is to control the way the batches are created and also to handle the batch size itself in order not to include, for example, a horizontal and a vertical image or a low-resolution and a high resolution image at the same batch, since we will have extreme padding and according to our hypothesis this can have a negative impact. Thus, large buckets may include big variety of shapes and sizes and small buckets won't allow to create different batches and may contribute for the model to overfit. Note also, that TPA is worst case scenario metric, thus, even if the TPA of a bucket is big, in a appropriate sized bucket, the TPA of a batch (a subset of a bucket) can be lower. Only when the largest and the smallest image within a bucket are batched together (something that may happen rarely, if ever), the padding will be maximum.

At the same time we need to maintain the stochasticity of the batching creation procedure as mentioned above. To do that we have to also disregard small buckets (buckets with a few number of images assigned to them) because then either almost the same batches will be created in all the epochs or a full batch can not even been formed. Like that we designed the bucketing approach as a pre-processing stage before training which is described in detail later in this chapter.

2.2 Bucketing

The goal of this part is to combine the right objective function with the right algorithm in order to minimize the padding area which is expressed through TPA eq.(2.1). The number of clusters $\mathcal{K}$ is something that it will be examined in the experiment in the next Chapter3.

2.2.1 Objective Functions

In *METavlitó* we used different objective functions(OF) in order to find the one that best describes the problem, the TPA minimization. For the needs of our use case we used 3 different functions that look at the problem of the padding minimization from different perspectives. The objective functions in clustering express also the dissimilarity metric based on which the clusters are formed.

- **Euclidean distance**

  This is a classic distance used as dissimilarity metric which is the "ordinary" straight-line distance between two points in Euclidean space. With this distance, Euclidean space becomes a metric space. With this we can align all images in a Cartesian space, as depicted in *Figure 2.3*. The objective function is the same *eq.(5.1)* as described in Appendix in *Chapter 5*. We write it again here following the same terms which were introduced in the previous section.

$$\mathcal{J} = \sum_{j=1}^{n} \sum_{k=1}^{K} w_{jk} \cdot \left\| i_j - \mu_k \right\|^2, \tag{2.3}$$

  where μ_k is the centroid or the dimensions of an image.

- **Sum of Euclidean distances**

  Using the euclidean distance as objective function we minimize the distance of 2 points, where the points are the upper right corners of the images when we place them in the Cartesian space as seen in the *Figure 2.3*. This could make sense since we want to find clusters with points as close as possible.

  Taking this example from the figure we can see that the Im2 has the same distance d to Im1 as to Im3 and then it will be assigned randomly to any of these to group with. For that we will investigate further what is should be done in these cases. Before we move on we need to simplify first *eq.(2.2)* for the case of 2 images like the ones in the *Figure 2.3*:

$$\mathcal{PC}(Im_i, Im_j) = prod(max(Im_i, Im_j)[x, y]) - prod(min(Im_i, Im_j)[x, y])), \tag{2.4}$$

  where $max(Im_i, Im_j)[x, y]$ is the max x and the maximum y of the images Im_i, Im_j.

  The *eq.(2.4)* actually gets the area of the big image and substracts it from the area of the small one in order to find the padding cost. So,

$$(2.4) \Rightarrow \mathcal{PC}(Im_i, Im_j) = max(Area(Im_i), Area(Im_j)) - min(Area(Im_i), Area(Im_j)) \tag{2.5}$$

  Specifically,

  $\mathcal{PC}(Im1, Im2) = Area(Im3) - Area(Im2) = wh - (w-1)(h-1) = w + h - 1$

  $\mathcal{PC}(Im3, Im2) = Area(Im2) - Area(Im1) = (w+1)(h+1) - wh = w + h + 1$

  So, $\mathcal{PC}(Im1, Im2) < \mathcal{PC}(Im3, Im2)$, which means that whenever the euclidean distance of a point is the same to another 2, it should group together with the one that's closer to the beginning of the axis $O(0, 0)$. In the case of our example, the TPA will be smaller if Im2 will join Im1 rather than Im3 since $d1 < d3$. Like that we constructed our own objective function which penalizes more bigger distances and also follows the same concept of the objective function of

euclidean distance in order not to change the calculations either in Expectation or Maximization step, as they are described in the Appendix in *Chapter 5* when we talked about the Expectation-Maximization Clustering.

$$\mathcal{J} = \sum_{j=1}^{n} \sum_{k=1}^{K} w_{jk} \cdot \left(\left\| i_j - \mu_k \right\|^2 + \left\| i_j \right\| + \left\| \mu_k \right\| \right), \tag{2.6}$$

where μ_k is the centroid or the dimensions of any image in the dataset.

- **Custom Padding Cost (PC)**

  Another way to address this minimization problem is through the areas of the images as it's implied by the different colors in *Figure 2.3*. For this we will work directly with *eq.(2.1)* and because the function is more complex we will work with different approach, trying to express the difference of the TPA when an element is added to a group and find the group that contributes less to the TPA.

  Specifically, when an image is added to a cluster:

$$PC(b_j, i_{new}) = (|b_j| + 1) \cdot prod(max(b_j, i_{new})[x, y]) - \sum_{\forall i \epsilon b_j} prod(i[x, y]) - prod(i_{new}[x, y]) \tag{2.7}$$

  So, from *eq.(2.2) & (2.7)* $\Rightarrow$

$$\mathcal{J} = \sum_{j=1}^{n} \sum_{j=1}^{K} w_{jk} \cdot \left(\mathcal{PC}(b_j, i_{new}) - \mathcal{PC}(b_j) \right) \tag{2.8}$$

  $(2.8) \Rightarrow \mathcal{J} = (|b_j| + 1) \cdot prod(max(b_j, i_{new})[x, y]) - |b_j| \cdot prod(max(b_j)[x, y]) - prod(i_{new}[x, y])$

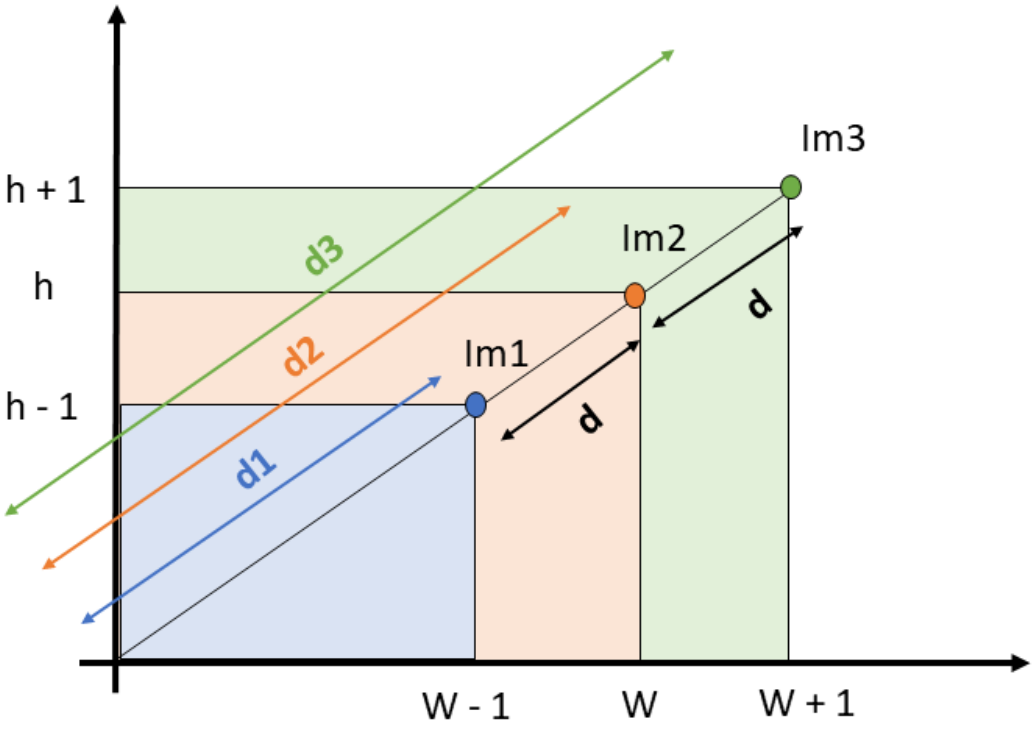


FIGURE 2.3: Cartesian space showing possible problems for different objective functions

2.2.2 Algorithms

For each of the above objective functions we implemented from scratch several clustering algorithms in order to find the one that can better address our padding minimization problem. From the analysis of the problem KMeans-based clustering algorithms seem to describe better the problem because of their simplicity in combination with the peculiarity of the metric (TPA) that we set to evaluate the results; however, we want to see other approaches how they can handle this use case. The chosen algorithms have the desirable property of allowing us to see how performance in TPA changes as the number of buckets changes; since different number of buckets affect differently the formation of batches during the training stage.

The algorithms that we implemented are:

- **Algorithm for finding number of clusters**

  One of the problems that may arise is the number of clusters that needs to be pre-defined. For that we implemented an algorithm called *Improved K-means Algorithm Based on Density Canopy*[38] and it will be used as a baseline algorithm. The algorithm of Density Canopy is used as the pre-processing procedure of KMeans and its result is used as the cluster number and the initial clustering centers of K-means algorithm. We extended this approach also for KMedoids, so we have:

  - Density Canopy KMeans
  - Density Canopy KMedoids

This algorithm try to improve the accuracy and stability of Kmeans algorithm and solve the problem of determining the most appropriate number $\mathcal{K}$ of clusters and best initial seeds. It calculates:

- the density of sample data sets
- the average sample distance in clusters
- the inter-cluster distance between clusters

In our case we used the distances we defined above. The sampling point with the maximum density is chosen as the first cluster center and it's removed then from the data sets. The paper[38] defines the metric of the weight product which is related with the product of

- the sample density
- the reciprocal of the average distance between the samples in the cluster
- the distance between the clusters

Finally, the other initial seeds are determined by the maximum weight product in the remaining data sets until the data sets is empty.

- **KMeans clustering algorithms variations**

  The approach of these algorithms solves the problem that is called Expectation-Maximization. The E-step assigns the data points to the closest cluster. The M-step computes the centroids/representatives of each cluster. The algorithms we implemented are different variations of KMeans, tweaking some parts and also changing the way they are initialized to limit the randomization step of this first stage of the algorithms.

  - KMeans
  - KMeans++
  - KMedoids (PAM)

  One of the things that this implementation adds to the standard algorithm is the way to handle the empty clusters. There are 2 options:

  - split the cluster with the highest error, given by the objective function
  - set as new centroid the instance that is the furthest away from the centroids

  In *METavlitó* the second method seems to be better, especially for the cases of many clusters (big $\mathcal{K}$) because otherwise the largest images contribute a lot in the TPA since the TPA depends on the maximum dimensions of each cluster/bucket.

  Furthermore, since the objective function is changed the E-step & M-step have to be re-calculated; the objective function of the euclidean distance is the classic KMeans which is analyzed in Appendix in *Chapter 5*.

  - *Define cluster formation and centroids for the Sum of Euclidean Distances*
    So, it's a minimization problem of two parts. We first minimize $\mathcal{J}$ w.r.t. w_{jk} and treat μ_k fixed and update cluster assignments (E-step). Then we

minimize $\mathcal{J}$ w.r.t. μ_k and treat w_{jk} fixed and recompute the centroids (M-step). Therefore, for the Sum of euclidean distances, E-step is:

$$(2.6) \Rightarrow \frac{\partial \mathcal{J}}{\partial w_{jk}} = \sum_{j=1}^{n} \sum_{k=1}^{K} \left(\left\| i_j - \mu_k \right\|^2 + \left\| i_j \right\| + \left\| \mu_k \right\| \right) = 0 \tag{2.9}$$

For solving this we will break it into 2 parts because both are above zero. The first part which comes from eq.2.9 has the following solution:

$$w_{jk} = \begin{cases} 1, & k = arg\min_l \left(\left\| i_j - \mu_l \right\|^2 + \left\| i_j \right\| + \left\| \mu_l \right\| \right) \\ 0, & \text{otherwise} \end{cases} \tag{2.10}$$

Concerning the M-step we want to find the right centroids:

$$\frac{\partial \mathcal{J}}{\partial \mu_k} = 2 \cdot \sum_{j=1}^{n} \left(w_{jk} \cdot (i_j - \mu_k + 1) \right) \tag{2.11}$$

The small constant that's added in the distance when we differentiate ends up as a constant added in the centroid and it is a residue from the penalization we added. However, in the centroids calculation it doesn't give any value as we can see. It only implies a very small shift towards the direction of the new added image's dimensions, but for the sake of simplicity we will ignore it. So,

$$\frac{\partial \mathcal{J}}{\partial \mu_k} = 0 \Rightarrow \mu_k = \frac{\sum_{j=1}^{n} w_{jk} \cdot i_{jk}}{\sum_{j=1}^{n} w_{jk}} \tag{2.12}$$

Which is the exactly same centroid as with the case of the euclidean distance described in Chapter5. That was also the thought behind the inspiration the Sum of Euclidean Distances objective function as it was mentioned above, to be as simple as the classic euclidean distance but overcome some of its shortcomings because of our use case.

– *Define cluster formation and centroids for the Padding Cost*
For this objective function we worked differently since we already had determined implicitly the centroid. The maximum of the dinstances of the cluster is already part of the objective function in *eq.(2.8)* representing the bucket as it's shown in *eq.*2.2. Also, since we calculate the difference of padding between an initial bucket (which means that it could also contain 1 image) and another new image we imply again that we want the minimum value of the objection function in order to assign an image to a cluster/bucket.

– *Define cluster formation and centroids for KMedoids*
KMedoids forms the clusters the same way KMeans does but the centroids are determined according to are the member of the cluster with the minimum sum of squared distance. This can work for the Euclidean and the Sum of Euclidean but not for the Padding Cost because depends heavily on the maximum dimensions of the cluster's elements as well as its size and thus, it's very computationally inefficient. So, following the same concept as before the centroids for Padding Cost will be the points

with the maximum dimensions but in this case they have to be points which exist in the cluster. To approximate this we calculate the maximum area, where $area = width * height$ and we assign as new centroid the point of the cluster with the maximum area.

- Hierarchical algorithms

  For the sake of completion we also included in this research project different ways to cluster as hierarchical clustering.

  - Agglomerative using Single Linkage
  - Agglomerative using Complete Linkage

  We worked with a distance matrix which had the pair distances. We use pair distances using Padding Cost as well without taking into account the size of the cluster, because of the complexity and the intensive computation of agglomerative algorithms in combination with the complexity of the Padding Cost equation eq.(2.8) that needs to find for each iteration the maximum dimensions between the formed clusters (note that the agglomerative algorithm is a bottom-up algorithm where each instance is considered a single cluster in the beginning).

- Custom algorithm

  This approach was inspired on genetic algorithms. A generic description of the clustering objective is to maximize homogeneity within each cluster while maximizing heterogeneity among different clusters, in a way that objects that belong to the same cluster are more similar than objects that belong to different clusters. However, a pure genetic solution is inefficient and infeasible for a large number of instances because of the time of the convergence; thus, it doesn't make sense for this implementation design given our use case. However, our approach is to refine clusters using the TPA as objective function (eq.(2.8) trying to re-allocate images in different clusters. With this idea in mind we define the Random Padding Refinement (RPR) Clustering.

  RPR implements the following steps:

  - initializes the very first single-image-buckets either randomly or according to a distribution like in KMeans++
  - shuffles the order of the images in each iteration
  - assigns the images to the clusters/buckets one-by-one that contribute less to the TPA. In other words, adding the image to a bucket or re-assigning it to another bucket decreases the TPA.
  - since each bucket grows bigger after assigning each image, the largest dimensions are calculated and their corresponding PC
  - when all images are assigned, they are shuffled again and they revisit all the new formed clusters in order to find to which bucket it has to be assigned in order to contribute less to the TPA
  - again after each image the maximum and the PC is calculated
  - this refinement process is continued until the TPA doesn't change and it reaches to a convergence

  The pseudocode looks like this:

```
TPA_previous <- 0
TPA_current <- 0
tolerance <- 0.001
initiliaze_buckets
while abs(TPA_previous - TPA_current) > tolerance
    TPA_previous <- TPA_current
    Ss <- shuffle the order of the images in each iteration

    for i ϵ Ss:
        l <- argmin_l (PC(b_l, i) − PC(b_l))
        if i ϵ b_j:
            b_j.remove(i)
        b_j_maxdim <- max_{x,y}(b_j)
        b_l.add(i)
        b_l_maxdim <- max_{x,y}(b_l)

    TPA_current <- TPA(B)
```

The shuffling is needed in order not to assign in each iteration the same images to the same buckets because then the process will finish immediately with a higher value of TPA. Like that we ensure that we don't depend on the order of the dataset and we can minimize the objective function as long as instances change from one bucket to the other. So, RPR can reach to a closer approximation of the best TPA given the initial buckets(intial points). The reason the images had to be parsed sequentially is because the eq.(2.8, 2.2) depends on the size of the bucket which changes every time an image is assigned to one.

2.3 Training

Training leverages on the buckets created in the previous stage. As first step here it is needed to prepare the images for training, through the pre-processing step. For that some specific dynamic transformation were employed to help our cause without interfering as much as possible with the way the network learns, using a specific architecture. The goal is to be able to make the network to learn feature maps of different images sizes (shape & scale).

2.3.1 Pre-processing options

- *Dynamic Resize/DownRatio*: It's the classic resize of the image but instead of setting constant dimensions, we introduce a ratio with which the image is resized with respect to the aspect ratio of the image in order not to deform it.
- *Dynamic Cropping*: We use this transformation as a data augmentation technique in order to make the network more resilient, and produce the same prediction even if the object is partly visible. This (hopefully) increases robustness against partial occlusions [32]. In our case we don't set again a constant size of the cropped image since the resized size of the image depends on its initial size. For that, we follow the same approach and we multiply the dimensions of each image with a pre-defined ratio. The bigger the ratio the more possibility not to fully crop out important parts of the image, which may lead to some unwanted false positives.

- *Normalization*: This is a classic step which ensures that each input parameter (pixel, in this case) has a similar data distribution. It makes the network to converge faster during training. The distribution of such data would resemble a Gaussian curve centered at zero.
- *Gamma Correction*: It performs a full color characterization according to the standard color correction pipeline. This was a more unconventional transformation that according to the literature [4] and also some experiments [10], it can be used for increasing the data or also for intensifying patterns and thus the activations later in the training. In general, this is able to normalize the images with respect to the color of the illumination.
- *Fourier Transformation*: It performs a transformation that can expand accessible information about the analyzed sample [5]. This was used as an alternative transformation of the representation of the image.

2.3.2 Bucketing Technique

In *Figure 2.4* we present the idea behind the training part of *METavlitó* project.

- According to the *Clustering* part above we split the dataset into a specified number of buckets.
- Each bucket consists of a number of images with similar sizes, which minimizes the *eq.(2.1)*. We need to disregard the buckets with low number of members/images otherwise we won't be able to create very different batches, losing stochasticity and generalization capacity
- Then, it's the turn of the Input Pipeline *Figure 2.4*:
  - A random bucket is picked
  - A number of images equal to the batch size is picked randomly in each step of each epoch. The batch size can be dynamic, which means that it can change and be bigger for buckets with smaller images and lower for images which include high resolution images. Like that it's possible to accelerate the training by maximizing the use of the GPU memory.
  - The specified transformations are applied. There can be a plethora of different ones from the classic one to more unconventional ones that affect the representation of the image. However, we chose to modify some of the classic ones in order to match the prerequisites we set and to help towards validating our assumptions without affecting a lot the representation of the input.
  - Padding is applied to each batch so that every image in there to match the maximum size of the dimensions of all the images in the batch, as it's depicted in the *Figure 2.4* below. This step is essential in order to avoid any size mismatch error during training. The smaller the difference in the size of the image the less padding is needed.
- The above steps end up being the final input of the CNN which is trained with a specific architecture for each given batch.

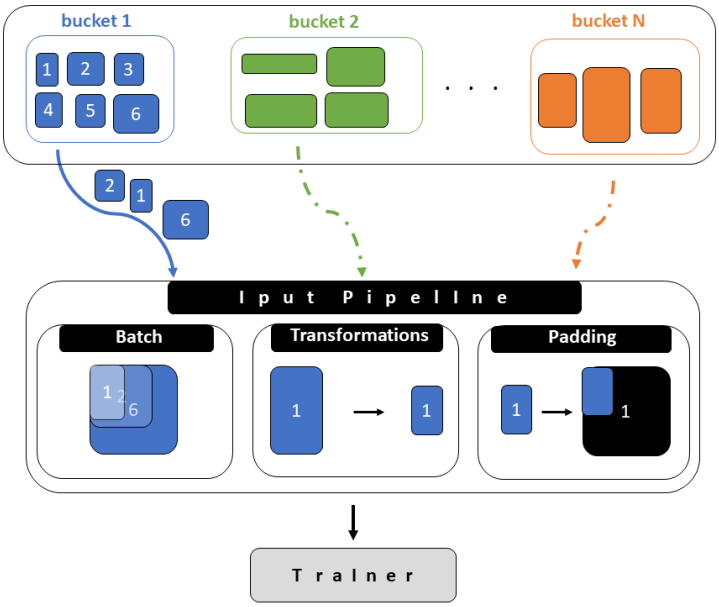


FIGURE 2.4: Bucketing technique design

Chapter 3

Experiments

This research project was built incrementally making some hypothesis, testing them and then building on top of them. As in [28] the various facets of the CNN architecture were kept constant while the pre-processing module was varied for different alternative approaches following the hypothesis presented in *Chapter1* in the *Hypothesis Space* section.

3.1 Dataset

The dataset used for experimentation is obtained from the open image and data resources of the Metropolitan Museum of Art in New York [**met-resources**]. The High Performance Artificial Intelligence (HPAI) group of the Barcelona Supercomputing Center (BSC) collected this data and created labeled datasets for different tasks [**meth-dataset**].

The specific dataset we are using for these experiments is the Medium dataset (MetH-Medium) *Figure 3.1*. This targets the identification of the medium a piece of art is made of (i.e., the main material of the piece). The dataset has 23 different classes, including Gold, Silver, Woodcut, Limestone, Silk and others.

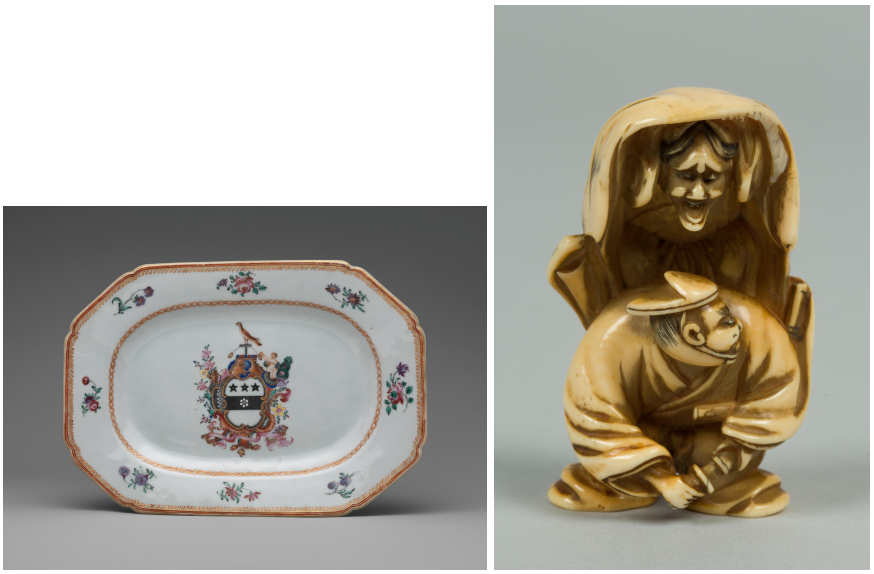


FIGURE 3.1: Example images of the Medium MET dataset

All the images are balanced across the 23 classes, each containing 1,000 images for training, 100 images for validation and 100 images for test. That means that the entire dataset contains a total of 27,600 images:

- 23000 images for training with 1000 instances per class
- 2300 images for both validation with 100 instances per class
- 2300 images for both testing with 100 instances per class

This dataset has all the characteristics that we want to experiment with in order to evaluate our hypothesis:

- big range of variable shaped images. As seen in *Figure 3.2* most of the images are horizontal (the aspect ratio is below 1) with some extreme cases of horizontal and vertical panoramas like the ones in *Figure 1.2*.
- big proportion of high resolution images. As seen in *Figure 3.2* most of the pictures are in HR and there are some which even surpass 4K definition.

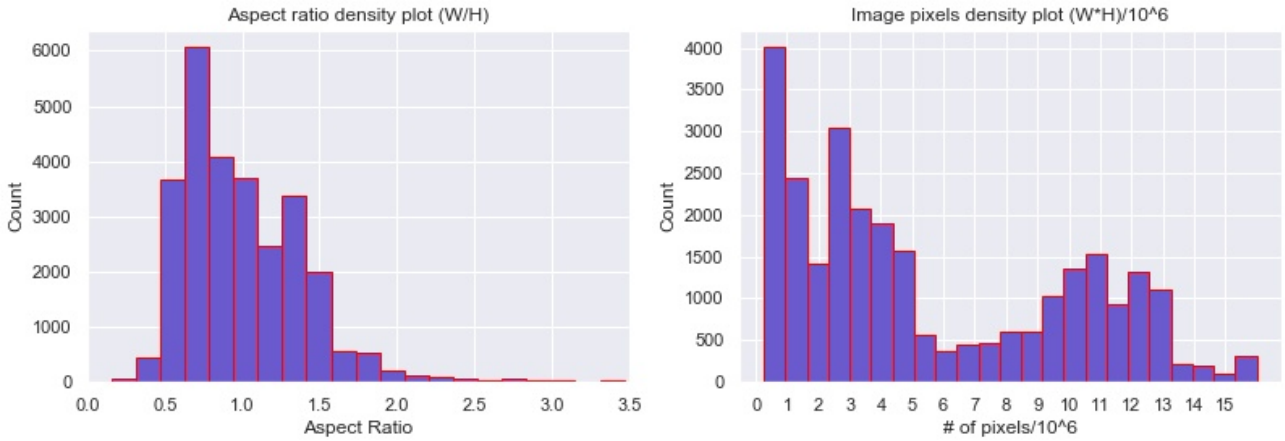


FIGURE 3.2: Distribution plots of the aspect ratio and the number of pixels to show the variance in shape and the number of HR images

Furthermore, there are 10 classes within the dataset which are quite hard to discriminate, even for humans like soft or hard types of porcelain (*Figure 3.3*). In these cases HR images may provide more information to distinguish the right class.

FIGURE 3.3: Samples of hard-paste, soft-paste and generic porcelain.

3.2 Bucketing Experiments

As first experiment we want to test our research on the objective functions and our implemented algorithms. What we want from these experiments is to measure

which objective function and algorithm minimizes the TPA. We will also consider their computational time, to make sure that the found solution is practical.

The KMeans-based algorithms contain some random operations when they are initialized and for that reason we ran the experiments 3 times and here we present the results of the average of these 3 runs.

3.2.1 Configuration

In these experiments we set the number of clusters K for the experiments in the range $[3, 20]$ in order to see the progression of the value of TPA as well as the time it takes. The reason behind this range is to manage to have as big clusters as possible where we can have different random batches with big batch size for the training which will be used in the experiments, as it was expained in the Chapter2.

The dimensions of the images in our dataset are between $[500, 6000]$, so the normalization is not necessary because one attribute is not disproportionately more important in its computation with respect to the other. However, the experiments were run with both configurations (normalized & non normalized) and here we present the ones without normalization after seeing that there was not significant difference. The maximum repetitions of each algorithm are set to 100 to let the clusters to converge especially for the experiments with high K. In addition, the tolerance set as a threshold for the convergence of the clusters is set to 0.001; if the difference of TPA or if the distance between the new centroid and the previous one of each cluster is less than the tolerance we assume that the algorithm has converged. Lower values of the tolerance did not change the results.

3.2.2 Objective Function Comparison

Here for each pair of objective function and algorithm we observe the progression of the TPA over the range of number of clusters K. At first, we will use as baseline the results from the Density Canopy which is used as a pre-processing step for KMeans and we also used it for KMedoids. We want to see the number of clusters discovered by the pre-processing and the TPA and time duration for each objective function.

Our baseline already gives us quite a few insights (*Figure 3.4*) of how our objective functions behave compared with each other. At first, apart from the euclidean objective function (OF) with which the Density Canopy finds 7 clusters for the rest it finds 6 clusters which is very close to each other. So, from that level we can see our custom objective functions work in terms of minimizing the defined disimilarity metric; even though the Density Canopy algorithm does not depend only on the distance, we can see that all the cases can find similar number of initial centroids which describe best the dataset, according to the Density Canopy algorithm.

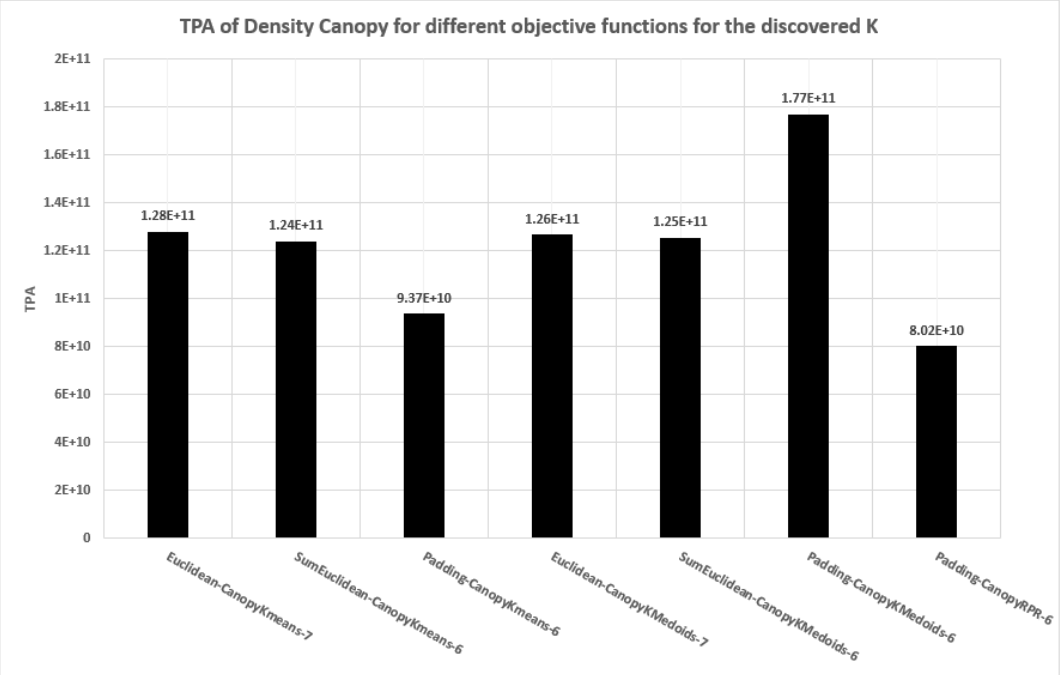


FIGURE 3.4: TPA of each objective function for the discovered number of clusters by Density Canopy. The number of clusters can be seen in the figure

Concerning the TPA which is used here as the error metric we can see an obvious dominance of our proposed Custom algorithm which uses the padding cost objective function with 6 clusters. KMedoids and KMeans at least for the discovered number of clusters have very similar results for the 2 euclidean based objective functions. However, padding objective function does not perform well with KMedoids while it gives much better results with KMeans. This has to do also with the way this objective function was designed and how it was used in KMedoids as it was described in Chapter2

After that we continued experimenting with how the different number of clusters effect the TPA for the different objective functions per algorithm. In *Figure 3.5* we can observe the same thing we saw above across all the different number of clusters. The padding cost based OF can minimize better the TPA since it is also derived directly from there. In addition, the sum of euclidean distance can actually give almost all the times slightly better results than the euclidean (in another scale the difference could be more apparent). The fluctuations mainly are due to the random initialization procedure.

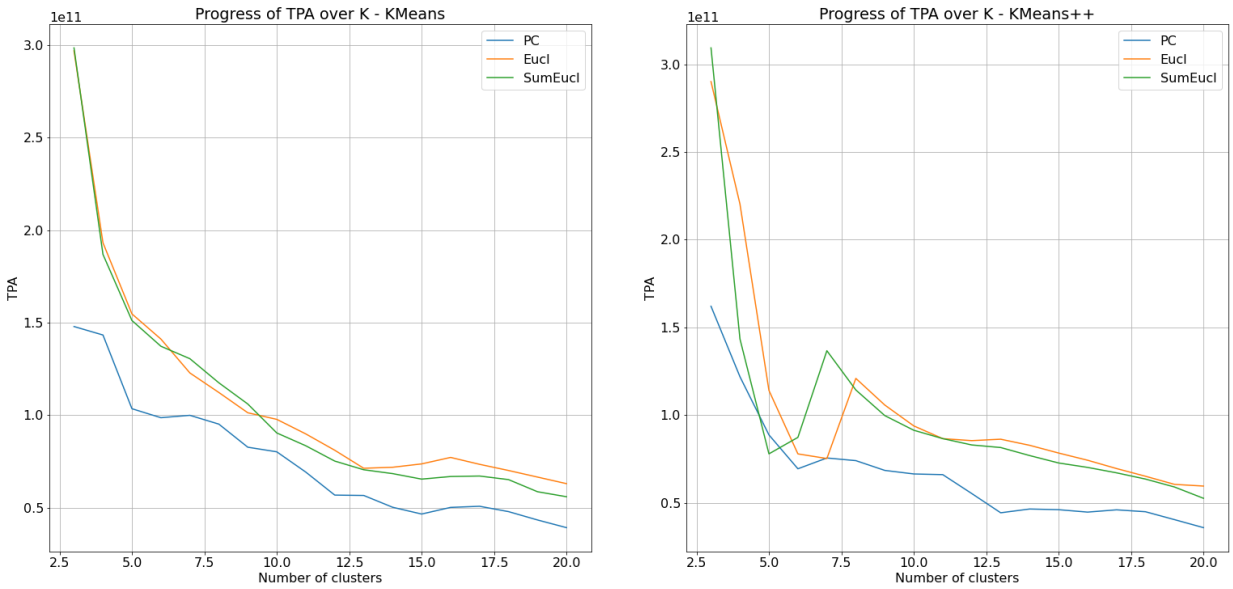


FIGURE 3.5: TPA over K for KMeans (left) & KMeans++ (right)

In the KMedoids we see different results *Figure 3.6*. The centroid assignment makes the 2 euclidean based objective functions to be close to each other over the different K. The Padding Cost OF cannot give the same results as above most probably because of the way the centroids are assigned. Nevertheless, for higher values of K the TPA reduces much faster and approaches the other results. The reason for this is that the clusters are more fine grained, thus, the way the new centroids are assigned make more sense but it can leave empty clusters throughout the training which are handled as mentioned in Chapter2.

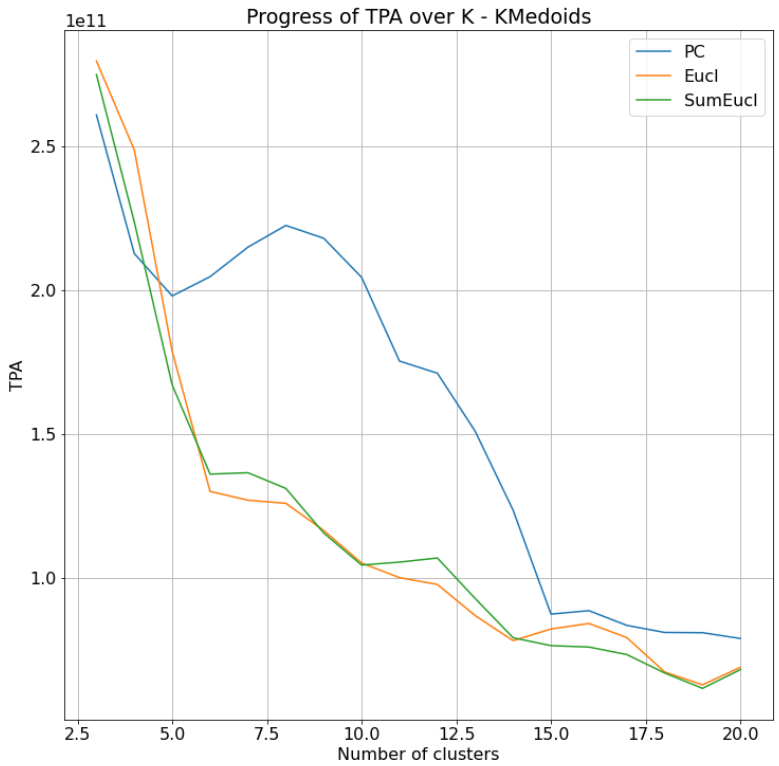


FIGURE 3.6: TPA over K for KMedoids

Finally, in the case of the Agglomerative algorithm we see again a bit different results. For the single linkage the 2 euclidean OF are a little bit better but after a certain K all OFs give almost the same results but because of the scale of the plot the result seem to be the same. Probably because of the density of the dataset (*Figure 3.14,3.15,3.16*) at a certain point the single linkage cannot really get informative clusters with many members probably because very close pairs are already found, which is what single linkage relies on. On the other hand, the complete linkage can work better and finds more distinctive clusters in comparison with the previous case because the TPA will depend on the largest pair between 2 clusters. Here since we work with pair distances the Padding Cost has lost big part of its advantage that is why in this case the Padding Cost gives the worst results. The Sum of Euclidean distances though, can again give better results than the Euclidean.

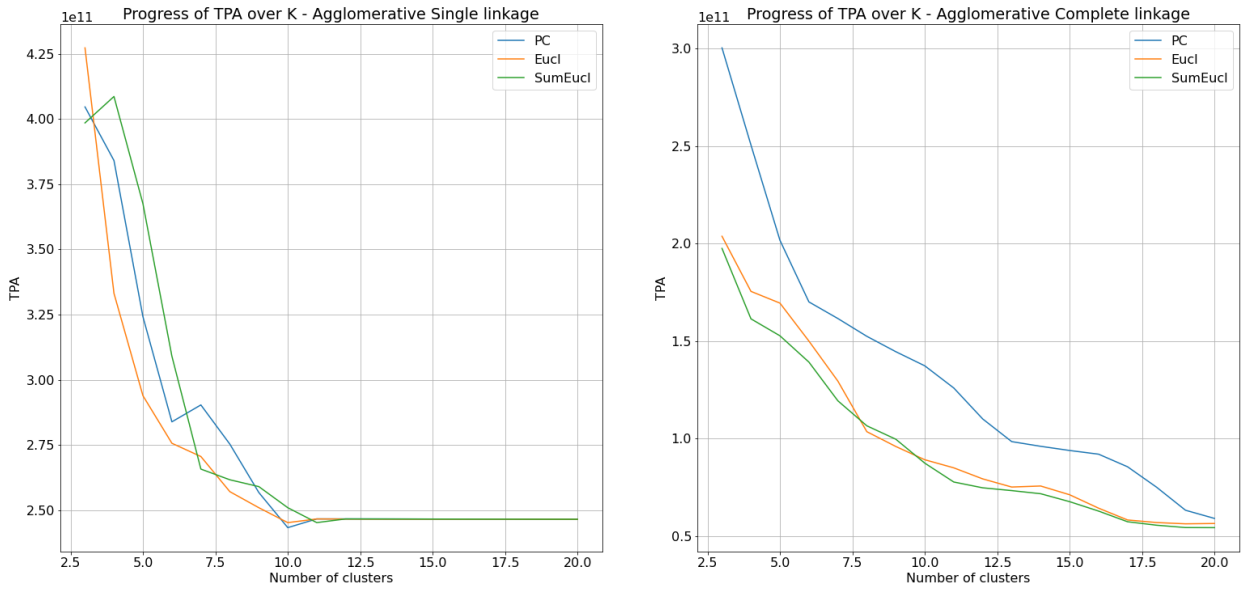


FIGURE 3.7: TPA over K for Agglomerative with Single Linkage (left) & Agglomerative with Complete (right)

In general, the TPA decreases as the number of clusters increase. Nevertheless, the baseline (given by the Density Canopy aglorithm) gives slightly better results for the specific number of clusters because the algorithms (KMeans & KMedoids) where initialized with more representative of the actual clusters centroids. So, given the number of clusters found from Density Canopy we see that RPR algorithm (our custom clustering implemention) gives better results.

Furthermore, concerning the way the TPA decreases, we notice that:

- in KMeans after $K = 8$ the TPA starts decreasing quite slower
- in KMedoids the decrease rate of TPA is lower after $K = 10$
- in Agglomerative it seems that the TPA's decrease rate does not change much even in the biggest number of clusters (20)

Finally, from the comparison of the OFs we can see the following:

- Sum of Euclidean performs better than the Euclidean OF
- Padding Cost outperforms both using KMeans & KMeans++
  - it gives worse results in the KMedoids and in Agglomerative (with complete linkage)
  - however, converges and approaches a lot (almost the same) the other 2 as K becomes bigger

Finally, in order to visually check the differences between OFs we present below the outcome of the clustering for KMeans with 6 clusters. As we can see from the *Figure 3.8,3.9,3.10* the different OFs provide different clusters using the same algorithm (KMeans). Specifically, the Sum of Euclideans seems to better separate the dense areas where there are many cases where an image can have the same distance

over 2 different clusters. The main difference between the 2 euclidean based OFs can be apparent in the upper right corner where there are many images with the same resolution and by splitting the clusters it achieved a lower TPA. On the other hand Padding cost did the same "trick" with the Sum of Euclidean distances for the large images, but it decided to have a big middle cluster for images between $2000 - 3000$ pixels and have a separate cluster for the outliers which have more than 4000 pixels in both dimensions which contribute a lot in the TPA. This can be an issue, since it does not guarantee stochasticity and as it was discussed in Chapter2 when we introduced the bucketing technique for first time, it should be handled appropriately. Note that TPA is a worst case scenario metric; in practice is going to be less, since the instances with maximum dimensions will not be used on all batches during the training stage, so a cluster which does not include just large images can create batches with less padding needed.

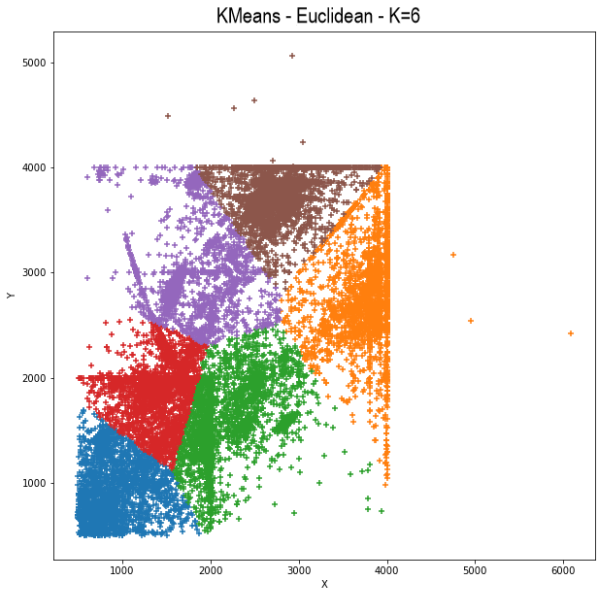


FIGURE 3.8: KMeans clustering using euclidean distance as OF with K=6

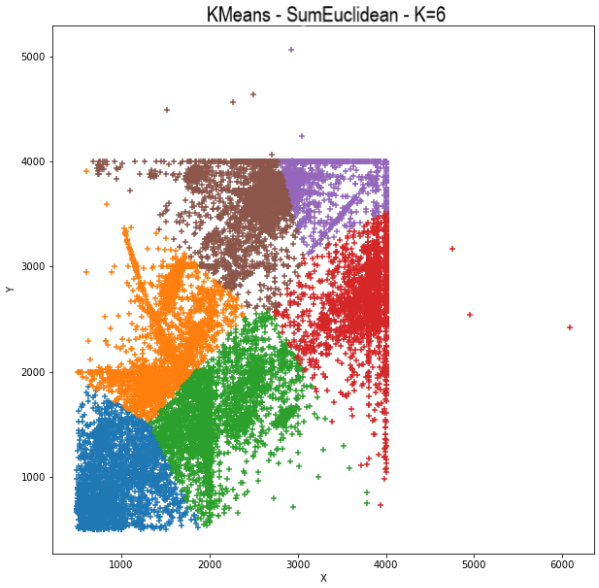


FIGURE 3.9: KMeans clustering using the sum of euclidean distance as OF with K=6

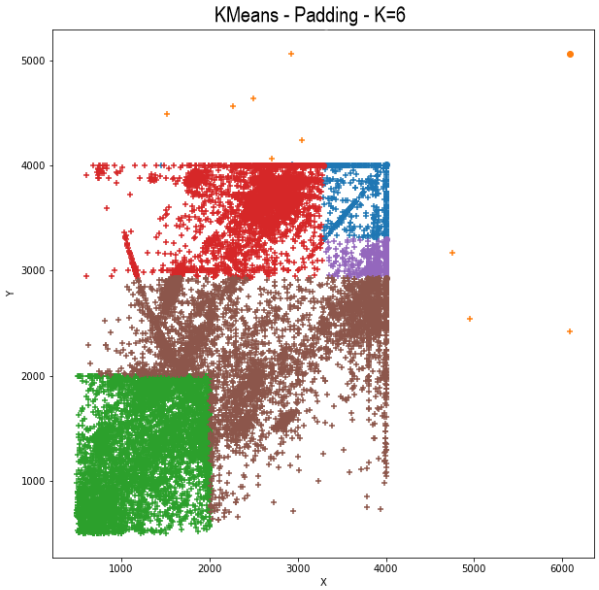


FIGURE 3.10: KMeans clustering using the padding cost as OF with K=6

3.2.3 Time & Algorithm Comparison

What follows is the comparison between the algorithms. Sarting again from the Density Canopy with the purpose to examine the time duration using KMeans & KMedoids for the discovered number of clusters. The duration seems to favor more

KMeans algorithm as depicted in *Figure 3.11*. This was expected from the theory because KMedoids has a much more complex way for calculating the new centroids. Concerning the OFs, the Euclidean OF seems to converge much faster than the others. The Sum of Euclidean follows next and the padding is the slowest; however that was also expected because it runs through all images sequentially to assign them in a cluster. The custom objective function seems to need much further time to converge which is mainly because of the refinement process of the clusters which is also the reason of having lower TPA.

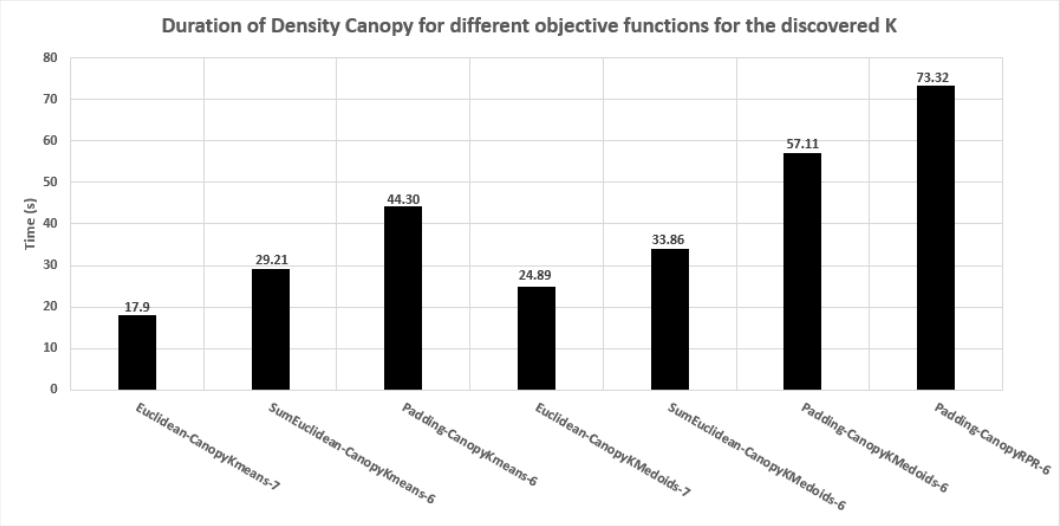


FIGURE 3.11: Duration of each objective function for the discovered number of clusters by Density Canopy

Since now we know which OF performs better for each algorithm we will compare the average time that each algorithm needs to converge to its final formed clusters. As depicted in *Figure 3.12* the results of the comparisons from Density Canopy seem appropriate and coherent. In addition, it seems like the Custom Algorithm does not change much its convergence time with more clusters probably because the number of iterations for a higher K are more or less similar as the ones with a lower K but with more elements going from one cluster to another, during the refinement process.

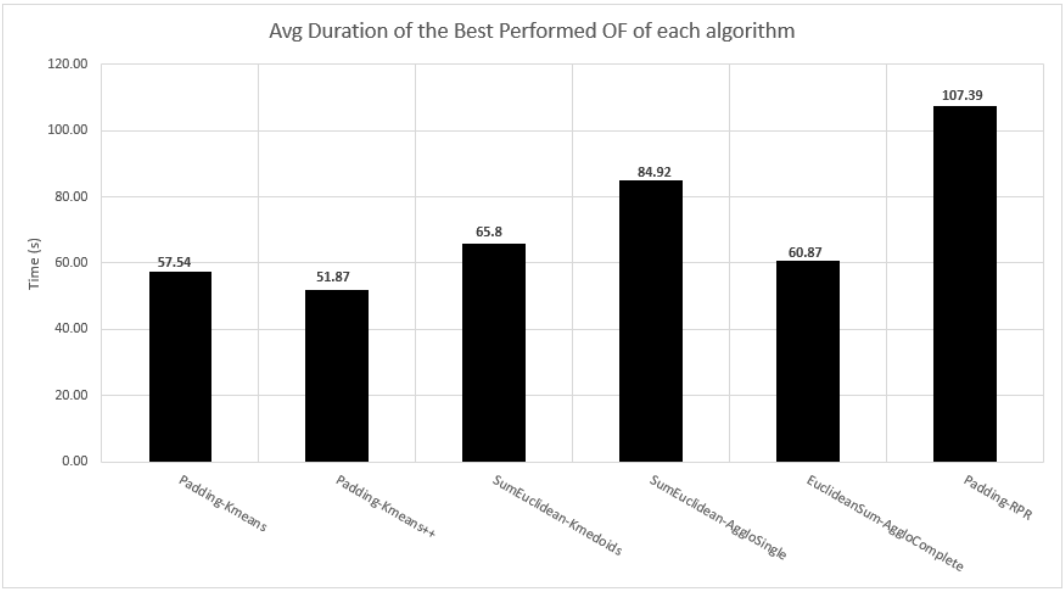


FIGURE 3.12: Average duration of the best performed OF for each algorithm

Overall, the time even of the slowest algorithm, our custom implementation (RPR), doesn't give a too much overhead since it takes less than 2 minutes to train. However, what is more interesting are the comparisons of the best results for each algorithm as seen in *Figure 3.13*.

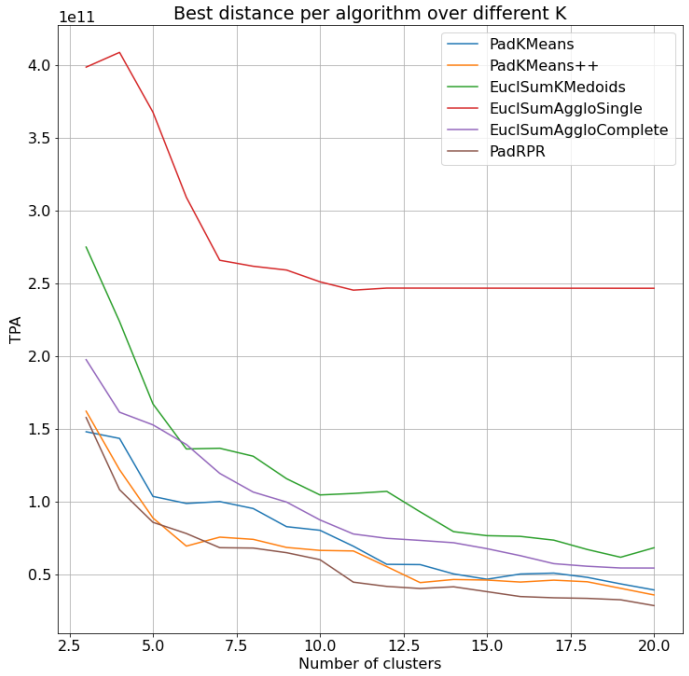


FIGURE 3.13: Comparison of TPA value between the algorithms using their best results from the OF comparison

As seen the best results are given using the Padding Cost as OF with our proposed RPR clustering algorithm and with KMeans++ being relatively close with the former being between 0-35% lower accross the different number of clusters. KMeans is approaching KMeans++ in higher values of K and Agglomerative algorithm using complete linkage follows with Sum of Euclidean distances as OF. Agglomerative using single linkage gives the worst results compared with the others by far.

Given the results above and since the TPA decreases over the number of clusters K we will use the results of our custom implemented algorithm *RPR* in the next component of *METavlitó* to experiment with different values of TPA.

Finally, the differences between the clustering algorithms using the same OF are not substantial visually, so, to see a more tangible example of how the different TPA looks like we present below 3 different clustering results of the dataset of images with $K = 6, 13, 20$ using our Custom Clustering Algorithm using Padding Cost as OF *Figure 3.14,3.15,3.16*.

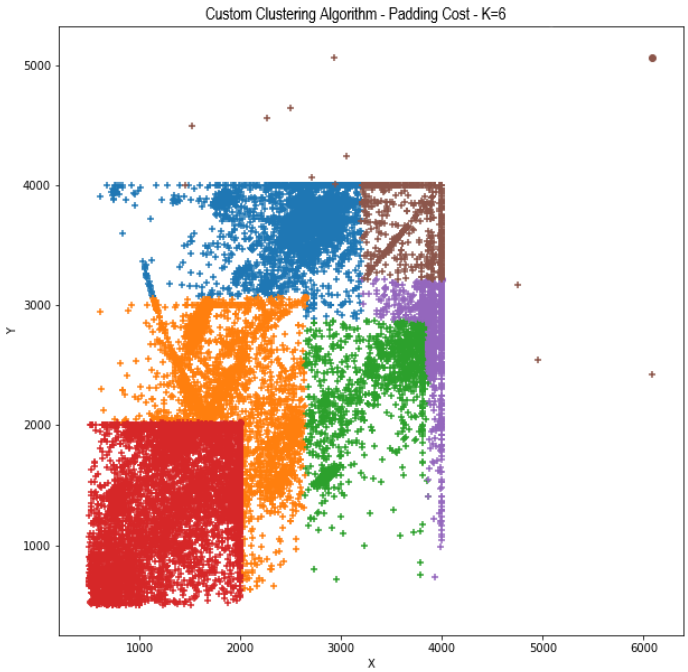


FIGURE 3.14: Clustering with K=6

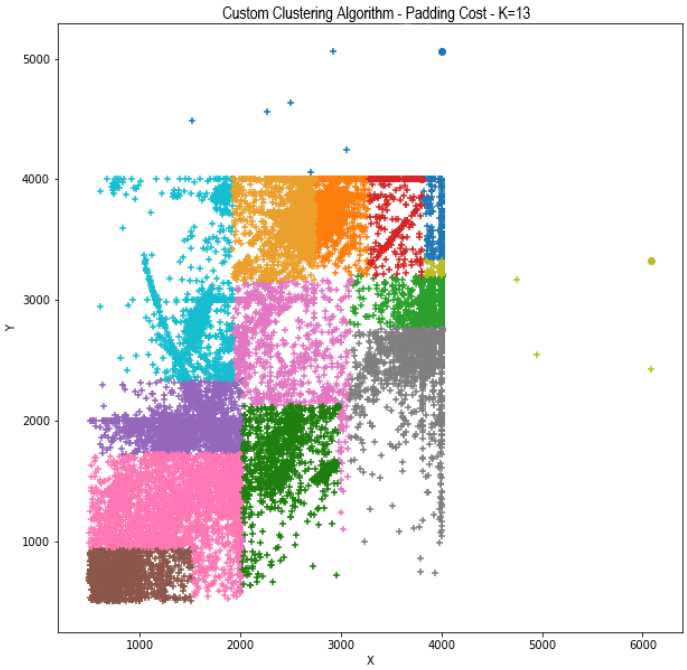


FIGURE 3.15: Clustering with K=13

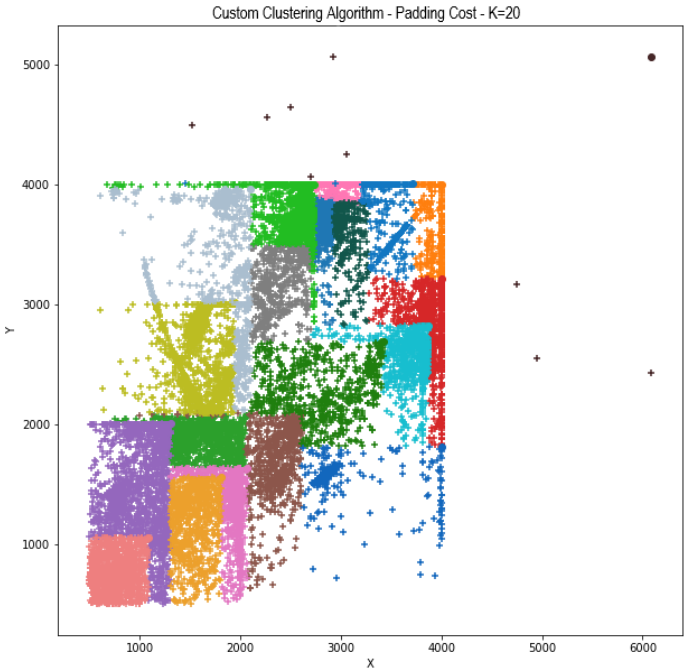


FIGURE 3.16: Clustering with K=20

3.3 Training Experiments

After finding a good candidate for bucketing of variable-shaped & HR images as input in a CNN, in this section we evaluate their impact on the training component to validate our hypothesis as they were described in Chapter1. *METavlitó* project effects the pre-processing stage of a CNN without interfering with the internals of how the network learns. For that reason we use one architecture in our experiments with specific initial parameters explained below. Having this set we continue with some experiments testing the proposed approach of *bucketing* and its possible advantages.

Our experiments were incremental building upon the results of the previous one, in a sense that we take 3-4 different pre-processing settings and we compare the results between them, the best result is the one we will continue with. The main key aspects to compare our experiments are the:

- Train & Validation Accuracy
- Train & Validation Loss
- Train & Validation duration per epoch (in seconds)
- Average Image Size per batch (in pixels) in order to see the computational overhead of using larger images given the other metrics. Also, this metric implies the padding needed at each batch, for example small average image size requires means that there was need of less padding
- Train & Validation Average GPU Utilization (%) in order to get a hint of the energy needed. Note that the utilization when the batch is loaded into the GPU is 100%, but what we measure is the average GPU utilization per step
- Train & Validation Average GPU Memory Used (MB) in order to get a hint of the GPU memory capacity needed
- Train & Validation Average CPU Load (%) we convert this into a percentage to see the backlog of jobs the CPU creates because of the pre-processing of the batches
- Train & Validation Average CPU Memory to see any potential needs in terms of CPU which is related also to pre-processing

3.3.1 Initial Parameters & Architecture

Before starting training we had to set some of the parameters of the network and of the pipeline we described before.

- We let the model to overfit in order to see all the progression of how it learns and it starts decreasing its capacity to generalize. For that we let the model learn for 20 epochs or 40 depending on the experiment and the baseline that the models are compared with.
- We validate the data at every epoch since theoretically we should also train our entire dataset at every epoch, each epoch yields:
  - $steps\_per\_epoch = TotalTrainingSamples/TrainingBatchSize$: we train exactly the entire training set

  - $validation\_steps = TotalvalidationSamples/ValidationBatchSize$, we validate exactly the entire validation set

- Concerning the optimizer, we used *Adam*, an adaptive gradient descent algorithm like *Adagrad*, *RMSprop* and others which provide an alternative to the classical *SGD*. These methods do not use the same learning rate for all parameter updates, but they provide heuristic approach without requiring expensive work in tuning hyperparameters. *Adam* adjusts the *Adagrad* method in a very simple way in an attempt to reduce its aggressive, monotonically decreasing learning rate.

  - The learning rate was set to $1e-4$ which gave some good results.
  - The learning rate decay over each update was correlated with the epochs like this: $learning\_rate/\#epochs$

- The transformations which will be finally used in the experiments from the ones mentioned in Chapter2 are the following:

  - Dynamic Resize/DownRatio: the ratio is changed according to the experiments
  - Dynamic Cropping: we let the ratio of the cropping dimensions constant to 0.875 because in most of the images of the dataset the information is in the middle of the image and the background doesn't provide any information (see *Figure 3.1*.
  - Normalization

  The rest of the transformations were discarded because they were not finally providing any added value and they were messing more with how the image is represented.

- Because of the different image sizes we want to experiment with, we need to change the batch size in order to fit in the memory of the GPUs. The batch sizes that are used throughout the experiments are $256, 104, 32$.

- We will experiment with $|B| = 6, 13, 20$ as options of different number of buckets. The idea behind these choices is to see the effect of different value of TPA when there is a big difference between them, like between 6 and 20 buckets or a smaller difference like between 13 and 20 where the TPA does not change as much (see *Figure 3.13*). As we can see from the images above (*Figure 3.14*,*3.15*,*3.16*) the outliers, the very large sizes of the images either group with the largest image in the top right corner as depicted in *Figure 3.14* or they are distributed among 2 *Figure 3.15* or 1 cluster *Figure 3.16* all together. In the latter case the cluster is formed by just 10 images which opposes to the requirements set in Chapter2 so we will need to try to include them in another cluster or to discard these images since they are not that many. In these experiments we decided to group them with another cluster. We remind here that images in a bucket are picked randomly for a batch to be created, so, the impact of including the very large images should be minimal since they are only 10 which means that can affect maximum 10 batches when the total amount of batches is more than 100 per epoch.

An important factor that affects the parameters above is also the capacity of the architecture, the depth and the length and so the complexity of the architecture

which adds more to the memory constraint in the training stage. The architecture chosen to experiment with is the *VGG16*. However, since our use case has to do with variable sizes we intervened in the architecture and in the last convolutional layer we replaced the Max Pooling layer which is the input to the fully connected layers with an Adaptive Average Pooling layer *Figure 3.17*, which depends on just the input and the expected output size of the layer.

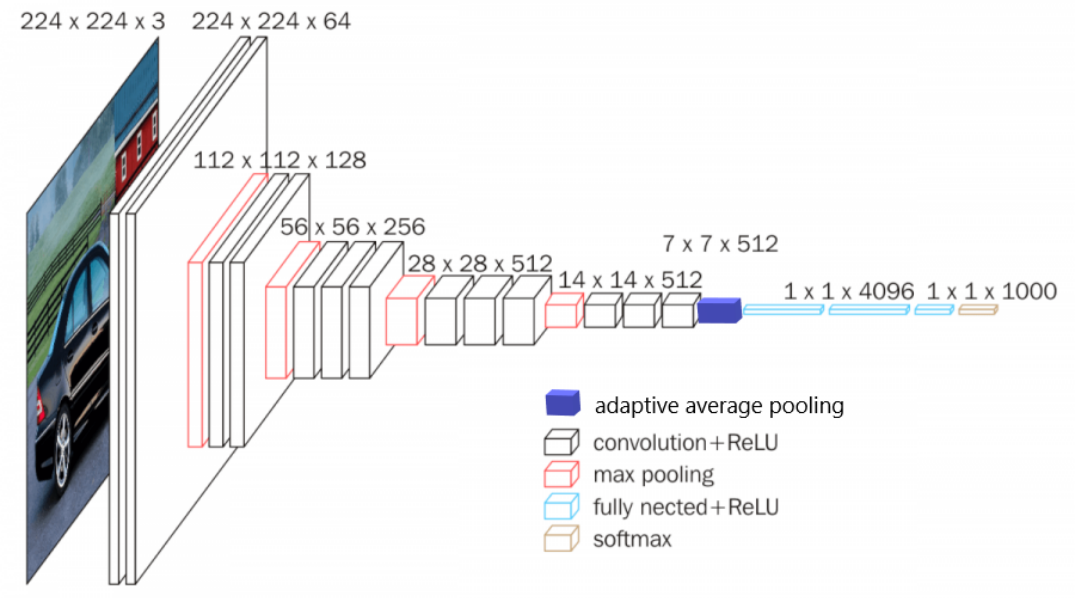


FIGURE 3.17: VGG-16 Architecture used in the experiments

3.3.2 Hypothesis #1: Loss of information

In the first experiment we want to evaluate Hypothesis 1; so when we want to see the effect of resizing high resolution images to a small one or to a bigger one. For this case we train all cases with the same batch size of 104 this number is defined by the memory needs of the most demanding case, where images are in size $400x400$. In *Table 3.1* there are the results of the runs resizing the images to $100, 256, 400$ and cropping them with $ratio = 0.875$, so we will refer directly to the dimensions of the cropped image. Here we use the model with the $256x256$ images as baseline.

Information Loss Experiments BS=104	Resize(256, 256) Crop(224, 224)	Resize(100, 100) Crop(80, 80)	Resize(400, 400) Crop(350, 350)
Train / Val Accuracy (%)	95.2 / 63.7	93.4 / 63.1	94.3 / **66.2**
Train / Val Loss	0.14 / 1.89	0.21 / 1.71	**0.17** / **1.62**
Train / Val Duration per epoch (s)	244.5	**163.5**	383.8
AvgImageSize per batch (pixels)	150528	**19200**	367500

Train / Val AvgUtilization GPU (%)	60.89 / 44.27	**47.36** / **32.14**	60.02 / 42.12
Train / Val AvgMemUsed GPU (MB)	22,350.25 / 11,986.25	**10,192.81** / **7,469.89**	38,226.06 / 14,692.06
Train / Val AvgLoad CPU (%)	20.54 / 14.16	25.37 / 20.44	**11.77** / **6.79**
Train / Val AvgMemUsed CPU (MB)	6,084.06 / 6,003.06	6,002.46 / 5,978.41	**3,180.87** / **3,017.81**

TABLE 3.1: Results from the loss of information experiment

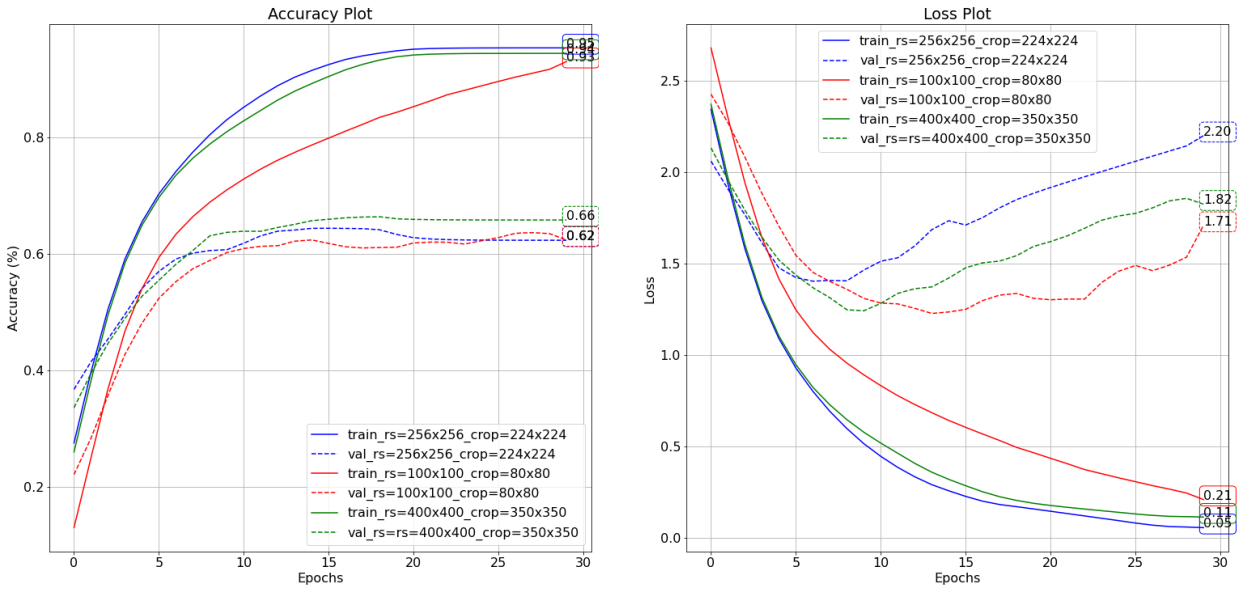


FIGURE 3.18: Loss of Info Experiment: Accuracy & Loss over the epochs

According to our hypothesis in Chapter1 loss of information can cause a decrease in accuracy. This is supported by this experiment, where the validation accuracy for the biggest resolution is the highest. However, results are not entirely conclusive, as the same does not happen with the loss the network learns more difficult with the $100x100$ resized images as it can also be seen in the Figure 3.18.

When we resize the images to $100x100$ it needs more time to converge that is why we increased the number of epochs up to 30. All of the models overfit a lot as we can see however, the 3rd model, which resizes the image to $400x400$, as we can see arrives to 66% of accuracy with less loss than the one it has at the end, when it reaches the 30th epoch. Concerning the rest of the metrics we can see that the bigger the image the more time needed for each epoch to be completed as well as the image size per batch which reflect in some way the GPU needs in terms of memory and

power; even though the GPU utilization for the images of 256x256 and 400x400 is almost the same which shows that both utilize the GPU more intensively per step. The CPU load describes the amount of active tasks requesting CPU resources to perform an action, but since the images of each batch are loaded slower in CPU memory when the images are bigger then this does not let many batches to wait to be served by the CPU and then by the GPU.

Following these results we will increase even more the image size, but to do that we needed to reduce the batch size to 32. So, we increased the image size to 600x600 and 800x800 with cropped images of 525x525 and 700x700 respectively. In the Table 3.2 we can see that the 2 big sized images as input give almost the same results in terms of accuracy and loss but the one with smaller size is less resource-hungry. Because of the big dimensions it is easy to reach to overfit as we can see from the *Figure 3.19*. The baseline as expected needs less resources but also performs worse since as the size of the input increases so does the GPU memory and utilization. The CPU on the other hand seems to be quite low in resource needs. Note also that the 600x600 corresponds to the 20% of a 3000x3000 image and the 800x800 to the 30% of bit of smaller image. That is why later we will prefer to continue our experiments using ratio of 20%.

Information Loss Experiments BS=32	**Resize(256, 256) Crop(224, 224)**	**Resize(600, 600) Crop(525, 525)**	**Resize(800, 800) Crop(700 700)**
Train / Val Accuracy (%)	96.1 / 64.6	95.3 / 67.8	95.5 / **67.9**
Train / Val Loss	0.14 / 1.88	0.15 / 1.58	0.16 / **1.58**
Train / Val Duration per epoch (s)	**381.7**	967.9	1601.3
AvgImageSize per batch (pixels)	**150528**	826,875	1,470,000
Train / Val AvgUtilization GPU (%)	52.71 / **46.41**	64.71 / 70.50	**50.72** / 74.47
Train / Val MaxMemUsed GPU (MB)	**12538.33** / **8,188.06**	33,908.6 / 14,286.0	55,758.4 / 19,560.0
Train / Val AvgLoad CPU (%)	9.02 / 6.34	4.61 / 2.66	**3.24** / **2.24**
Train / Val MaxMemUsed CPU (MB)	**3037.27** / **3009.00**	3,131.45 / 3,020.12	3,232.88 / 3,043.03

TABLE 3.2: Results from information loss experiments (batch size 32)

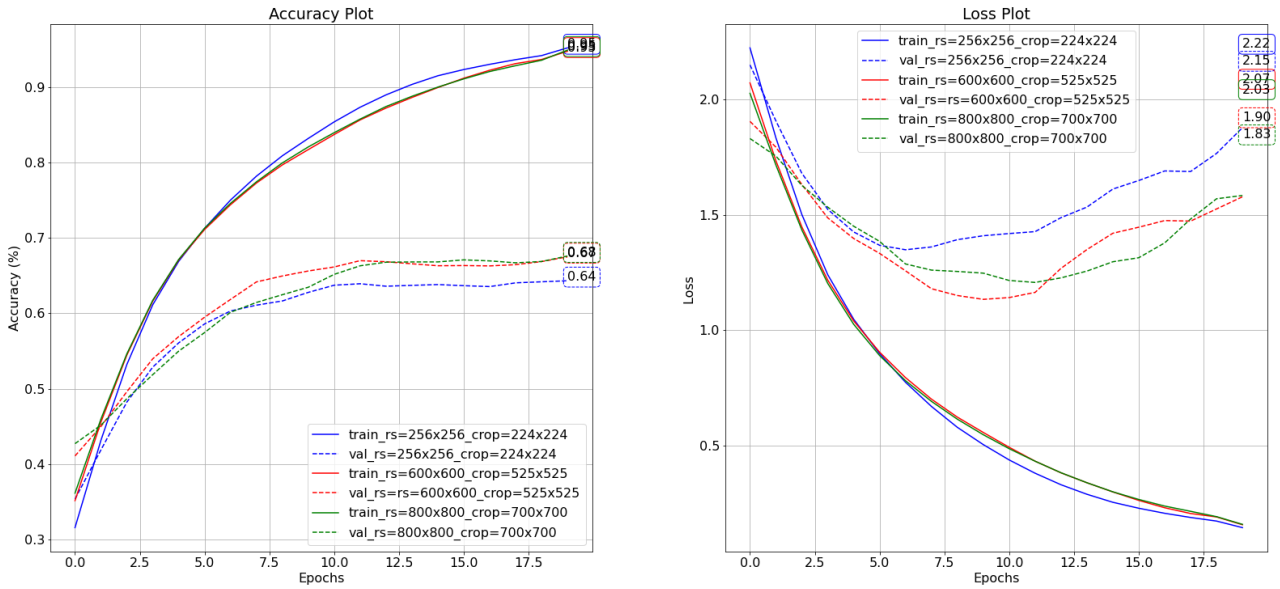


FIGURE 3.19: Loss of Info Experiment with bigger images: Accuracy & Loss over the epochs

3.3.3 Hypothesis #2: Shape Deformation

In the deformation experiments (see Hypothesis 2 we want to see the effects of changing more drastically the shape of the images in each batch using as baseline the model which uses images of dimensions 256x256. In these experiments we experiment with resizing images to:

- 1000x65 and then crop them to 900x55 making the images to stretch vertically increasing a bit the total area of the images of the baseline
- 100x1000 and then crop them to 50x900 in order to stretch this time the images horizontally and also decreasing this time a bit the total area of the image compared with the baseline
- 1300x60 and then crop them to 1200x40 which stretches more the images vertically and they they have almost the same area with the baseline

Deformation Experiments	Resize(256, 256) Crop(224, 224)	Resize(1000, 65) Crop(900, 55)	Resize(100, 1000) Crop(50, 900)	Resize(1300, 60) Crop(1200, 40)
Train / Val Accuracy (%)	95.29 / 62.3	93.75 / 67.82	84.8 / 67.61	93.75 / **67.91**
Train / Val Loss	0.14 / 1.86	0.18 / 1.11	**0.44** / **1.06**	0.28 / 1.14
Train / Val Duration per epoch (s)	247.3/ 27.3			
Image Size per batch (pixels)	150,528	148,500	**135,000**	144,000

Deformation Experiments	Resize(256, 256) Crop(224, 224)	Resize(1000, 65) Crop(900, 55)	Resize(100, 1000) Crop(50, 900)	Resize(1300, 60) Crop(1200, 40)
Train / Val AvgUtilization GPU (%)	**65.8/41.3**			
Train / Val AvgMemUsed GPU (MB)	19,350 / 9,986	19,352 / 9,942	**18,186 / 9,586**	18,924 / 9,850
Train / Val AvgLoad CPU (%)	20.54/ 14.16	**17.69 / 12.33**	19.58 / 13.5	16.48 / 11.54
Train / Val AvgMemUsed CPU (MB)	**6084.1 / 6007.1**			

TABLE 3.3: Results from the deformation experiments

The results from the Table 3.3 show quite unexpected compared with our hypothesis and apparently this is caused because of the nature of the problem which includes high-resolution images. We didn't actually stretch the images but we approached the actual size of the image (since most of the images are above 800x800).

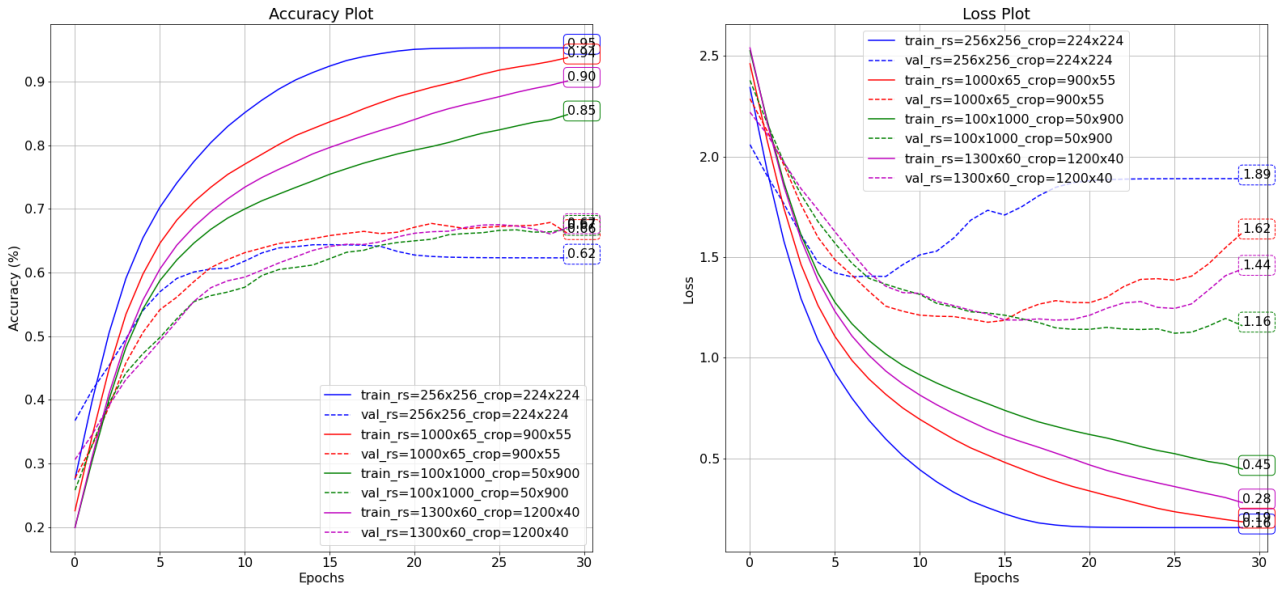


FIGURE 3.20: Deformation Experiment: Accuracy & Loss over the epochs

The results with images of 256x256 are very similar with the ones before but in this case we changed the batch size to 256 since now the images for all the experiment could fit in the GPU memory. The overfit though affected a bit more the accuracy of the model that is why it is 1% less.

In general, it looks like that the results we get depend on the dimensions they have and their aspect ratio. More specifically, we observe that bigger dimensions

give better results. Nevertheless, in the case of the wider image with cropped image size $50x900$ we obtain the best potentially results (less overfiting) and most probably because of the aspect ratio distribution of the dataset; there are more horizontal images than vertical. However, because it has the smallest area and dimension (which is reflected by its image size per batch and thus the GPU memory needs) the accuracy does not even reach 85% after 30 epochs but it reaches a surprising 67,61%, compared with the other results.

Even though it overfits more the model with the final cropped image size of $900x55$ provides slightly bigger images (in terms of the total number of pixels) and as we saw in the previous experiments this can give a potential advantage in terms of accuracy which we can see it here where it reaches 67.91%. The rest of the metrics are quite similar apart from the GPU memory as it was commented above.

An important gained insight that we did not anticipated was the regularization capacity of deformation. Even though the area provided as input in the training is similar in all the cases the deformation we applied affect the convergence of the models. It seems that this type of deformation works like a data augmentation technique since the training accuracy and loss (*Figure 3.21*) converge slower and it balances the variance and bias of the model in all the deformed cases. This means that for tasks where deformation may not be an issue (where the proportion of visual patterns is not important, like with this dataset), implementing it may be beneficial.

Another insight we gained is having HR in at least one axis is more important than avoiding deformation. The benefits of one compensates the inconveniences of the other. The problem is, we can not experiment with extreme deformation without causing big dimensions. In hindsight, we probably should have tested with medium deformations that did not result in HR in an axis. Thus, we can not reject the hypothesis and probably another dataset should be found for better exploring this area.

3.3.4 Hypothesis #3, #4: Padding

In this part of our work we want to experiment with padding (see Hypothesis 3, 4) as a way to group together images of different sizes. We want to test different paddings and the effect they have finally in the CNN's performance. For that we use bucketing, which was explained earlier, as a way to minimize padding and no bucketing to see the effects of extreme padding when we batch randomly together images of different size. So, we have no or extreme padding ($Buckets = 0$), big padding ($Buckets = 6$), medium padding ($Buckets = 13$), small padding ($Buckets = 20$).

- We use dynamic resize (downratio) by keeping its aspect ratio (downratio) and set the max size of the images in order to fit them in the GPU memory with batch size of 256. Like that most of the images will have similar sizes maintaining their aspect ratio.
  - The baseline is again the case with the normal resizing of the image to $256x256$ and crop it to $224x224$
  - We apply a down ratio of 10% setting the maximum image size to $256x256$ and then we crop with a ratio of 0.875 (which in the case of $256x256$ it gives again $224x224$). We use simple padding to batch the images together.
  - The same as above but we use the bucketing technique to batch the images together using specifically 6 buckets.

Padding Experiments BS=256	Resize(256, 256) Crop(224, 224)	DownRatio=0.1 MaxSize(256, 256) CropRatio=0.875 Buckets=0	DownRatio=0.1 MaxSize(256, 256) CropRatio=0.875 Buckets=6
Train / Val Accuracy (%)	94.0 / 63.4	95.0 / 64.44	91.1 / **65.2**
Train / Val Loss	0.17 / 1.76	0.14 / **1.70**	0.26 / 1.77
Train / Val Duration per epoch (s)	**204.12**	216.56	222.02
AvgImageSize per batch (pixels)	150,528	**148,430**	145,534
Train / Val AvgUtilization GPU (%)	58.34 / 46.98	57.41 / 46.62	**51.9** / **48.63**
Train / Val AvgMemUsed GPU (MB)	43,478.0 / 41,098.25	43,478.0 / 42,602.0	**40,748.0** / **40,108**
Train / Val AvgLoad CPU (%)	13.78 / 9.78	16.24 / 10.91	**11.02** / **7.41**
Train / Val AvgMemUsed CPU (MB)	**6165.23** / **6009.60**	6532.31 / 6106.15	6172.87 / 6010.94

TABLE 3.4: Results from the padding experiments (batch size 256)

The experiments supports the hypothesis. However, its not entirely conclusive because the results as expected are quite close to each other with a small 1% improvement in the accuracy in each case. As we can see the little added information from bucketing (using 6 buckets) contributes to a higher training loss and lower training accuracy while the validation is more or less the same, giving an advantage to this solution because it overfits less than the others, as seen in Table 3.4. That is also evident from the accuracy and loss curve shown below *Figure 3.21*.

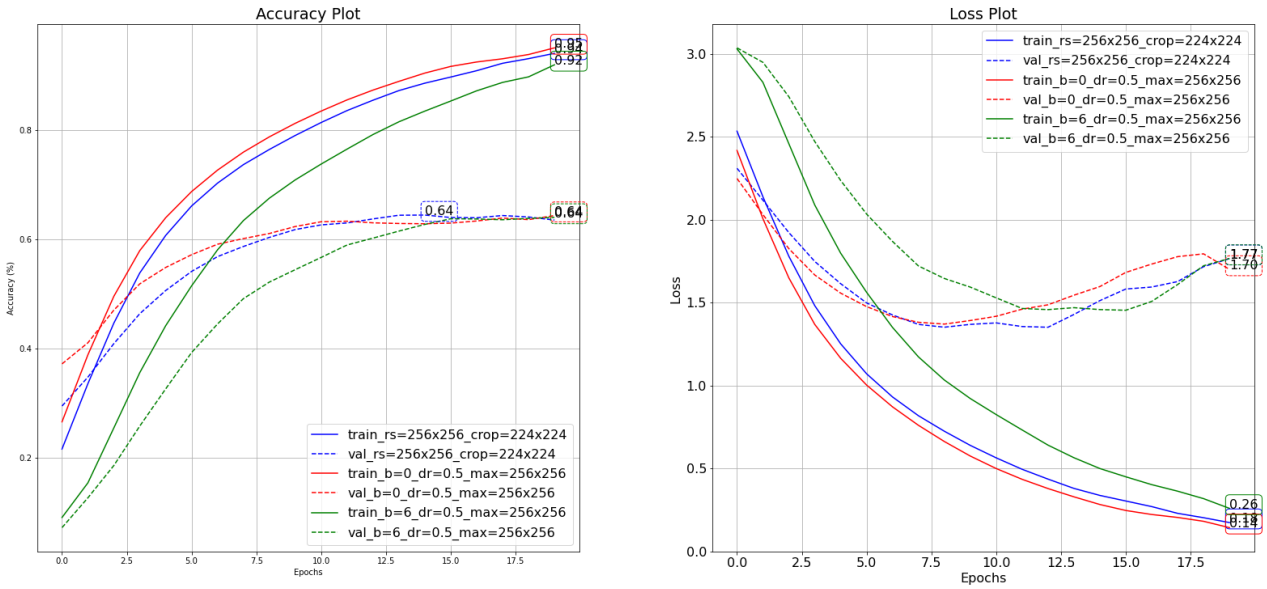


FIGURE 3.21: Padding experiment with max size set: Accuracy & Loss over the epochs

In our hypothesis we clarified that bigger padding should result in more memory needs in GPU. The padding is implied by the image size per batch and the memory needs from the average GPU memory. The correlation between these two in this case can be more obvious since the image size changes in every step; also, seeing the plots over the steps below we can start extracting conclusions. For presenting better the results without lines hiding behind fluctuations we smoothed a bit the lines over a window of 101 to limit the extreme fluctuation.

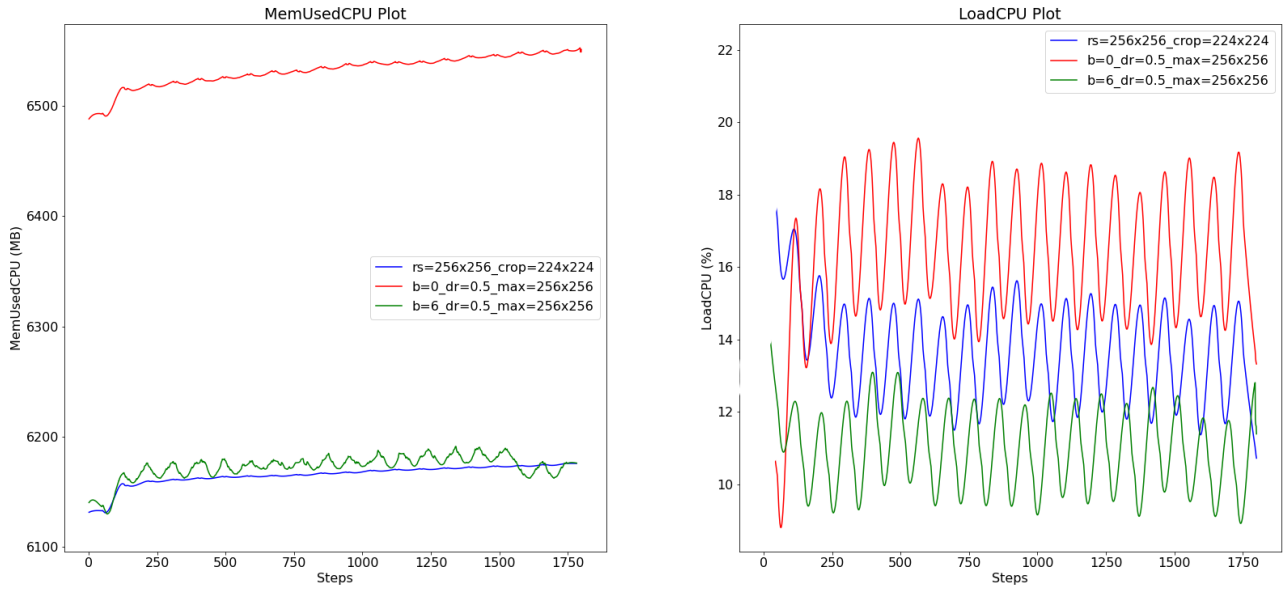


FIGURE 3.22: Padding experiment with max size set: CPU Load & Memory over steps (smooth fluctuations over a window of 101)

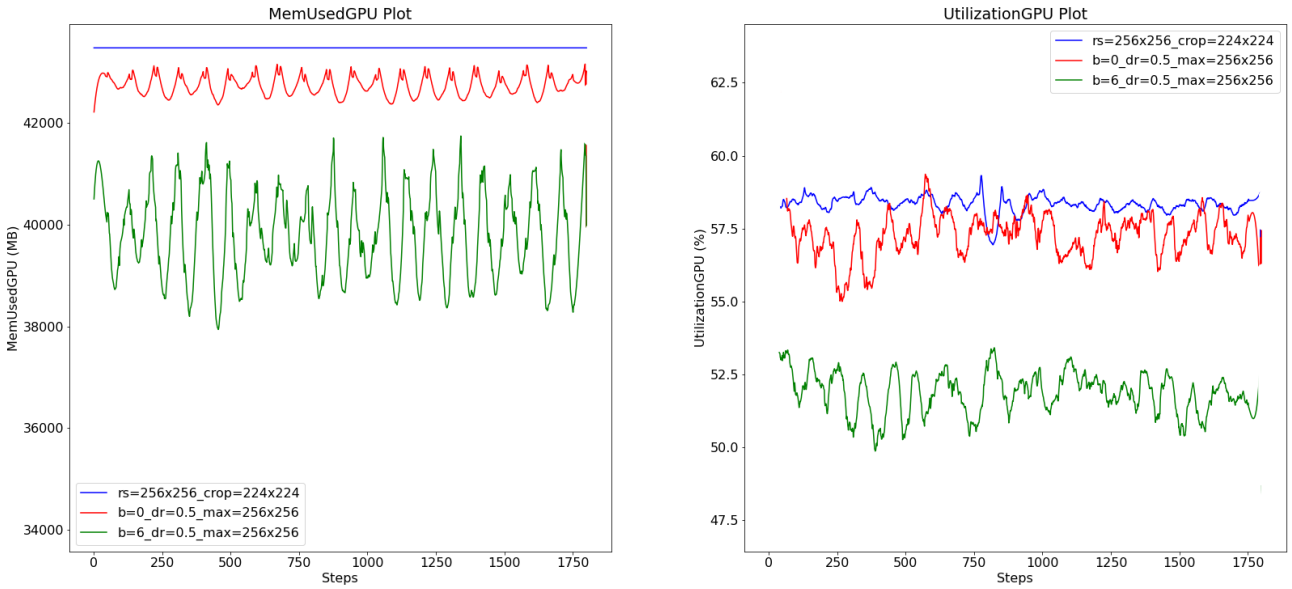


FIGURE 3.23: Padding experiment with max size set: GPU Utilization & Memory over steps (smooth fluctuations over a window of 101)

From the plots above in *Figures 3.22, 3.23* we can observe the following:

– In CPU memory there is very little difference between padding and no padding

- At the very beginning of the computation the CPU the memory is used quite heavily because of the image preparation adding padding to each of them to batch them together
- Less padding needs less CPU during data preparation and more padding needs more CPU during computation. With the bucketing technique we need to pad less so that is why we need less CPU memory compared without using it. However, the CPU memory needs are not big
- In the load of the CPUs we can see something different. Padding without bucketing is more computationally intensive but when we use bucketing we can see that the load in CPUs is less.
- Concerning the GPU we can see an apparent benefit over using bucketing. Less padding needs less GPU in average because it loads batches with smaller images. We need to note though that as we have set the same maximum size in all the cases here the real values of the GPU memory can fluctuate up to the baseline model.
- Finally, the average GPU usage over steps seems also to favor more the bucketing technique since in general can give more often batches with smaller images. Note that the GPU is measured over a step, from loading the images in GPU until the network updates its weights in a pass.

Padding Experiments Buckets BS=256	**DownRatio=0.1 CropRatio=0.875 Buckets=6**	**DownRatio=0.1 CropRatio=0.875 Buckets=13**	**DownRatio=0.1 CropRatio=0.875 Buckets=20**
Train / Val Accuracy (%)	91.1 / 65.2	96.1 / 65.9	96.3 / 66.7
Train / Val Loss	0.24 / 2.44	0.07 / 2.05	0.09 / **1.88**
Train / Val Duration per epoch (s)	222.2	215.7	**209.3**
AvgImageSize per batch (pixels)	145534	139363	**138041**
Train / Val AvgUtilization GPU (%)	59.91 / 49.63	52.8 / 40.61	**50.21** / **39.57**
Train / Val AvgMemUsed GPU (MB)	39,807.40 / 40,708.25	37,407.32 / 37,512.25	**35,587.23** / **34,476.06**
Train / Val AvgLoad CPU (%)	**10.98** / **7.41**	11.24 / 7.77	11.91 / 8.77
Train / Val AvgMemUsed CPU (MB)	**6,202.62** / **6,021.56**		

TABLE 3.5: Results from the padding experiments (batch size 256) using the bucketing technique with buckets=6, 13, 20

After observing the benefits of the bucketing technique we continue with comparing the different bucketings with $B = 6, 13, 20$. As we can see from the results in the *Table 3.5* we can see that with bigger number of buckets you can get better potentially results, since through the accuracy and the loss curves *Figure 3.24* we can see less overfitting and in terms of duration we manage to decrease it and approach the duration of the baseline. This validates our hypothesis3 that we can get better accuracy with less padding as well as that it needs more epochs to converge. This reinforces the idea of regularization through variable shaped images with little padding.

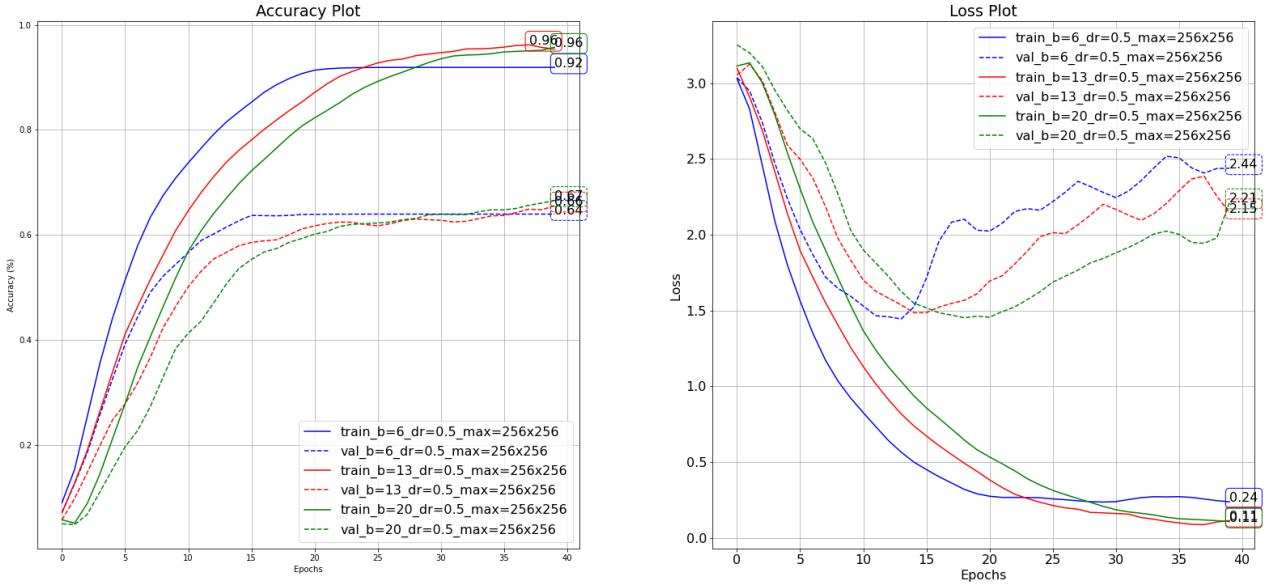


FIGURE 3.24: Padding experiment with max size set - Bucketing: Accuracy & Loss over epochs

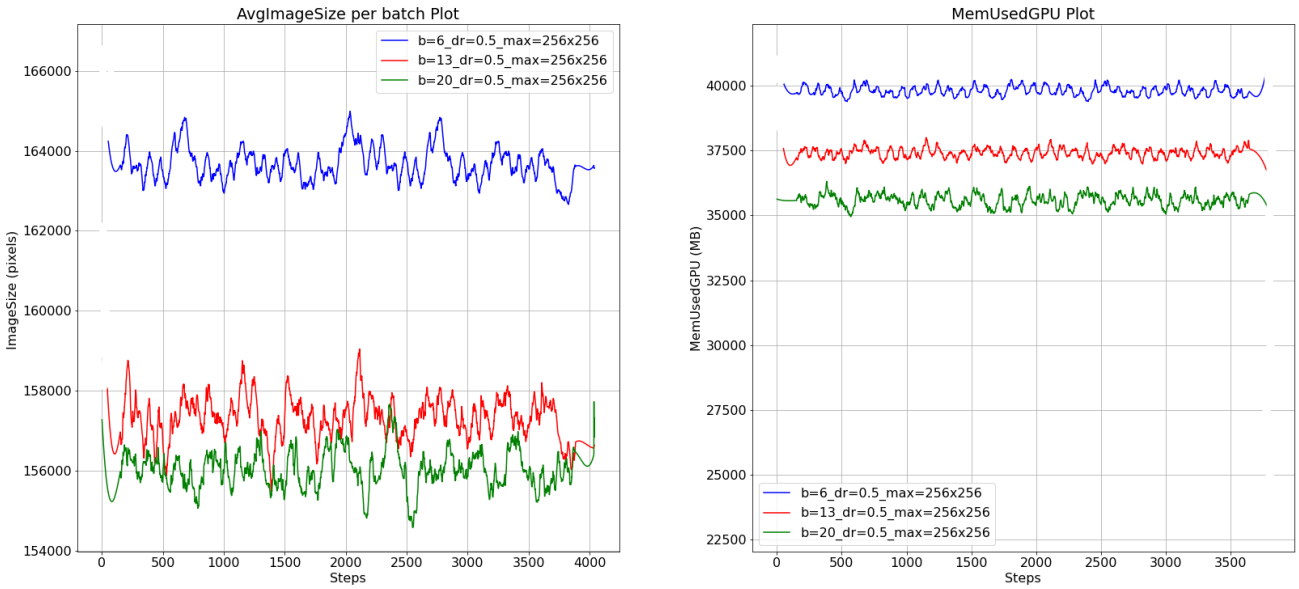


FIGURE 3.25: Padding experiment with max size set - Bucketing: Image sizes per batch & GPU Memory over steps

In the Accuracy & Loss plot in *Figure 3.24* we can see also that even though the final accuracy and loss of the largest number of buckets is better, it takes more time for it to converge. The reason behind this effect is the way we force the network to learn with variable sized images grouped in buckets which may result in more steps than in the other runs (because each bucket may not be divided exactly with the batch size). In general, in the whole bunch of experiments we conducted the bucketing could give $1-3\%$ more in accuracy.

Also, in this experiment we can get the information of the padding needed at each batch checking the average image sizes per batch. As we can see in the table above (Table 3.5) the more buckets we have the smaller the average image size and thus, the padding needed in each batch. We can also see that in the plot *Figure 3.25* that this links to a much better memory usage in average that is at the end $4GB$ less than the initial baseline.

- In this set of experiments we use downratio of 10%. Like that we let all images to be resized but we don't set the maximum size of the image as before (where we forced to have maximum an image of $256x256$). For that reason we decrease a bit the batch size for the experiments with larger images to fit in the memory. So, here we want to actually decrease the size of the images by 10% (with respect to the aspect ratio) and evaluate the performance of the CNN when we feed it with various input sizes.

  Our baseline for this set is again the same as before with resize of $256x256$ and cropped final image $224x224$. In the beginning we compare the baseline with:

  - resized image to $400x400$ and cropped image of $350x350$
  - we down ratio the image with ratio of 0.1 without using the bucketing technique

We can see the results in the Table 3.6, where the case with the $400x400$ resized image seems to give the best results. The non-bucketing technique can cause large image sizes per batch (extreme padding) with extra computing power and time needed to pad the much bigger images. So, here it seems that a bit of loss of information in the images gives better results than extreme padding.

It needs to be noted that the $400x400$ image size is close to the 10% of the largest image of the dataset. However, despite the benefits that padding can have, because the batches are created randomly we cannot really benefit from that in this case. This can be an evidence that extreme padding does decrease CNN's performance.

Padding Experiments BS=104	Resize(256, 256) Crop(224, 224)	Resize(400, 400) Crop(350, 350)	DownRatio=0.1 CropRatio=0.875 Buckets=0
Train / Val Accuracy (%)	95.2 / 63.7	**94.3** / **66.2**	92.2 / 64.4
Train / Val Loss	0.14 / 1.89	**0.17** / **1.62**	0.22 / 1.83
Train / Val Duration per epoch (s)	**244.5**	383.8	400.3
AvgImageSize per batch (pixels)	**150528**	367500	361704
Train / Val AvgUtilization GPU (%)	60.89 / 44.27	**60.02** / **42.12**	60.19 / 40.79
Train / Val AvgMemUsed GPU (MB)	**22350** / **11986**	38226 / 14692	38872 / 20087
Train / Val AvgLoad CPU (%)	20.54 / 14.16	**11.77** / **6.79**	15.66 / 8.46
Train / Val AvgMemUsed CPU (MB)	6084 / 6003	**3180** / **3017**	3265 / 3988

TABLE 3.6: Results from the padding experiments (batch size 104)

Since the experiment with the largest initial image $400x400$ gave better results we will continue with that to compare it with other number of buckets ($B = 6, 13, 20$) and any benefits we can get through this technique.

Padding Experiments BS=104	Resize(400, 400) Crop(350 350)	DownRatio=0.1 CropRatio=0.875 Buckets=6	DownRatio=0.1 CropRatio=0.875 Buckets=13	DownRatio=0.1 CropRatio=0.875 Buckets=20
Train / Val Accuracy (%)	94.3 / 66.2	90.5 / 66.8	89.18 / 67.8	**89.01** / **68.4**
Train / Val Loss	0.17 / 1.62	**0.33** / **1.13**	0.32 / 1.15	0.30 / 1.16

Padding Experiments BS=104	Resize(400, 400) Crop(350 350)	DownRatio=0.1 CropRatio=0.875 Buckets=6	DownRatio=0.1 CropRatio=0.875 Buckets=13	DownRatio=0.1 CropRatio=0.875 Buckets=20
Train / Val Duration per epoch (s)	383.8	325.4	299.7	**285.5**
AvgImageSize per batch (pixels)	367500	225692	198794	**180300**
Train / Val AvgUtilization GPU (%)	**60.02** / **40.12**	71.46 / 49.75	72.42 / 59.37	69.20 / 49.18
Train / Val AvgMemUsed GPU (MB)	38226 / 14692	25987 / 24557	23552 / 22031	**21494** / **22243**
Train / Val AvgLoad CPU (%)	**11.77** / **6.79**	15.85 / 10.17	16.61 / 10.93	17.14 / 11.56
Train / Val AvgMemUsed CPU (MB)	3180 / 3017	**2988** / **2988**	3205 / 3036	3229 / 3138

TABLE 3.7: Results from the padding experiments (batch size 104) resizing to a bigger scale of the image and using the bucketing technique with buckets=6, 13, 20

In Table 3.7 we can observe that adding more buckets provides an accuracy improvement from 13 to 20 that according to *Figure 3.13* gives results relatively close to each other. The loss seems to be very close between the models which use the buckets, but there is a difference in accuracy; however that was not the case all the time since we observed an improvement of $0 - 2\%$ over different runs. This can be seen also in the *Figure 3.26* where we see that the bucketing contributes to balance more the overfitting and that in accuracy the results are close to each other and fluctuate a bit.

In terms of time there is a quite big difference as we increase the number of buckets because of the smaller average image size per batch which in turn implies the total padding needed in each case *Figure 3.27*. Because the values of the memory of CPU don't seem to be a burden we present also the progress of memory over the steps during training. In both cases we see that the bigger number of buckets, the less needs for padding the less memory requirements during training. In addition, the utilization of GPU at this stage is more or less the same since it's quite heavily used using this kind of batch size with this kind of images. Finally, note again that through this bucketing technique throughout our experimentation we were getting between $3 - 5\%$ more in accuracy compared with the initial baseline of size $256x256$.

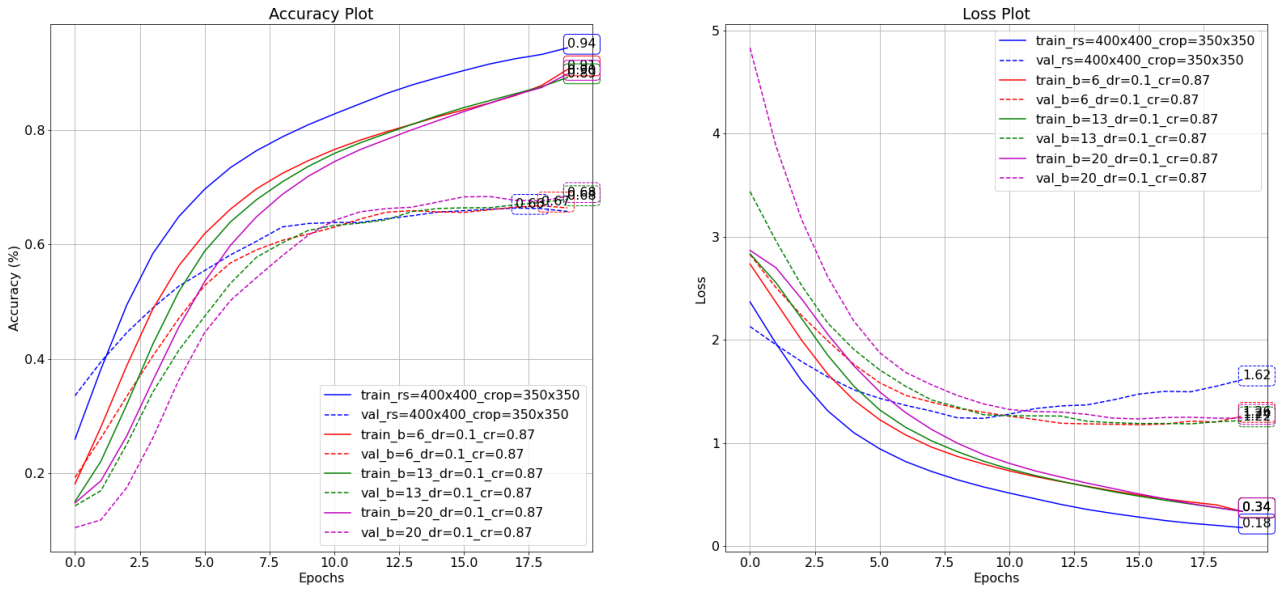


FIGURE 3.26: Padding experiment with 10% downratio - Bucketing: Accuracy & Loss over epochs

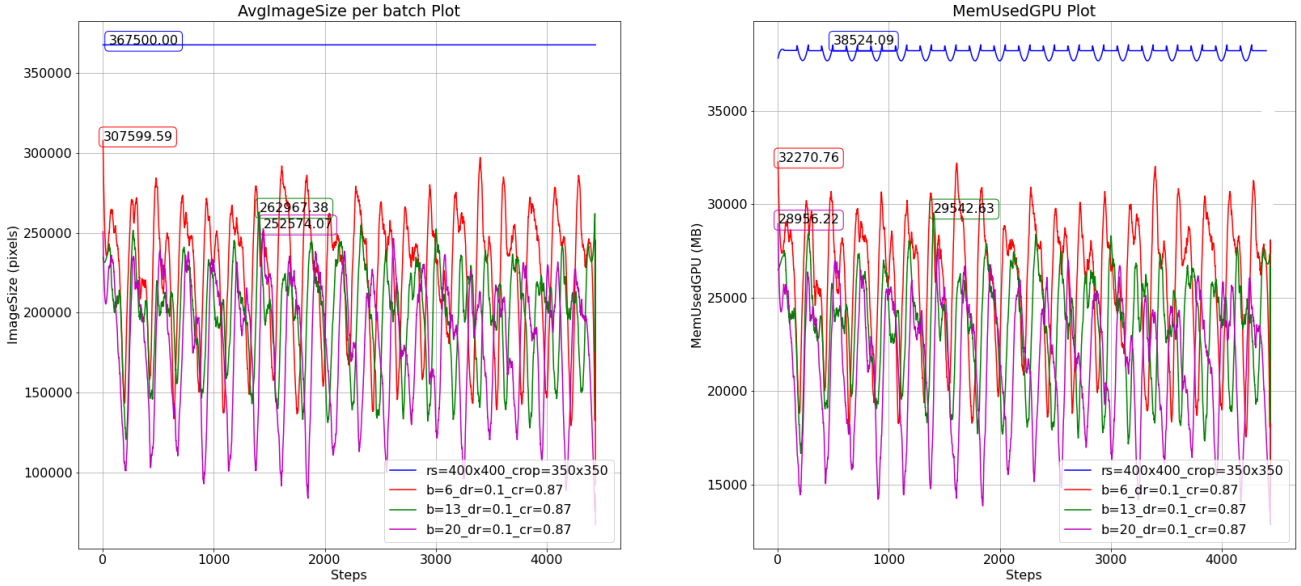


FIGURE 3.27: Padding experiment with 10% downratio - Bucketing: Image size per batch & GPU Memory over epochs

- Finally, we use down ratio of 20% without any thresholds on the sizes. That means also that we increase the image size which force us to reduce the batch size to 32. In this setup we used our conclusions from the previous experiments, so here we first compare with the baseline (with initial image size

256x256 and cropped size 224x224) the effect of even bigger images per batch. The results using padding without the bucketing technique were not comparable in this experiment and also as a solution didn't perform very well, especially compared with the models using bigger resized images. For that reason, we didn't include it in these results and at the same taking the insights from the results from loss of information experiments *Table 3.2* we used here as baseline the one with resized images of 600x600. Note that an image of 600x600 corresponds to the 20% resize of an image of 3000x3000, which is a bit smaller than the biggest one.

In Table 3.8 we can see here that the more information that the baseline includes gives it an advantage, however the results are quite close with each other in accuracy and loss. The advantage of the bucketing technique are is the decreasing amount of time needed to train the network as the number of buckets increases.

Padding Experiments BS=32	Resize(600,600) Crop(525,525)	DownRatio=0.2 CropRatio=0.875 Buckets=6	DownRatio=0.2 CropRatio=0.875 Buckets=13	DownRatio=0.2 CropRatio=0.875 Buckets=20
Train / Val Accuracy (%)	95.5 / 67.8	94.3 / 67.6	92.1 / 67.3	94.5 / **67.9**
Train / Val Loss	0.15 / 1.58	0.17 / 1.46	0.24 / **1.32**	**0.18** / **1.28**
Train / Val Duration per epoch (s)	967.9	905.2	830.6	**802.7**
AvgImageSize per batch (pixels)	826,875	738,072	645,132	**622234**
Train / Val AvgUtilization GPU (%)	64.71 / 70.50	66.25 / 56.17	64.86 / 50.75	**63.52** / **49.83**
Train / Val AvgMemUsed GPU (MB)	33,908 / 14,286	30,850 / 35,856	27,786 / 37,693	**26,890** / **36,246**
Train / Val AvgLoad CPU (%)	**4.61** / **2.66**	6.05 / 3.27	6.34 / 3.56	6.54 / 3.66
Train / Val AvgMemUsed CPU (MB)	**3,131.45** / **3020.12**	3,181 / 3,084	3,155 / 3,069	3,232 / 3,149

TABLE 3.8: Results from the padding experiments (batch size 32) using the bucketing technique with buckets=6, 13, 20

In addition, compared with the previous set of experiments (Figure 3.7) the results here are not improved. Specifically, the models now overfit more and don't show so much of an improvement in accuracy. Probably the reason would be the much smaller batch size which can affect the performance; that was also the reason that led to a lower learning rate according to [22],[6]. The results of bucketing are quite close even in the average padding needed (implied through the image size per batch) and the memory needs (see *Figure*

3.29. This is also depicted in the *Figure 3.28* where the differences throughout the steps is much smaller compared with the previous set of experiments.

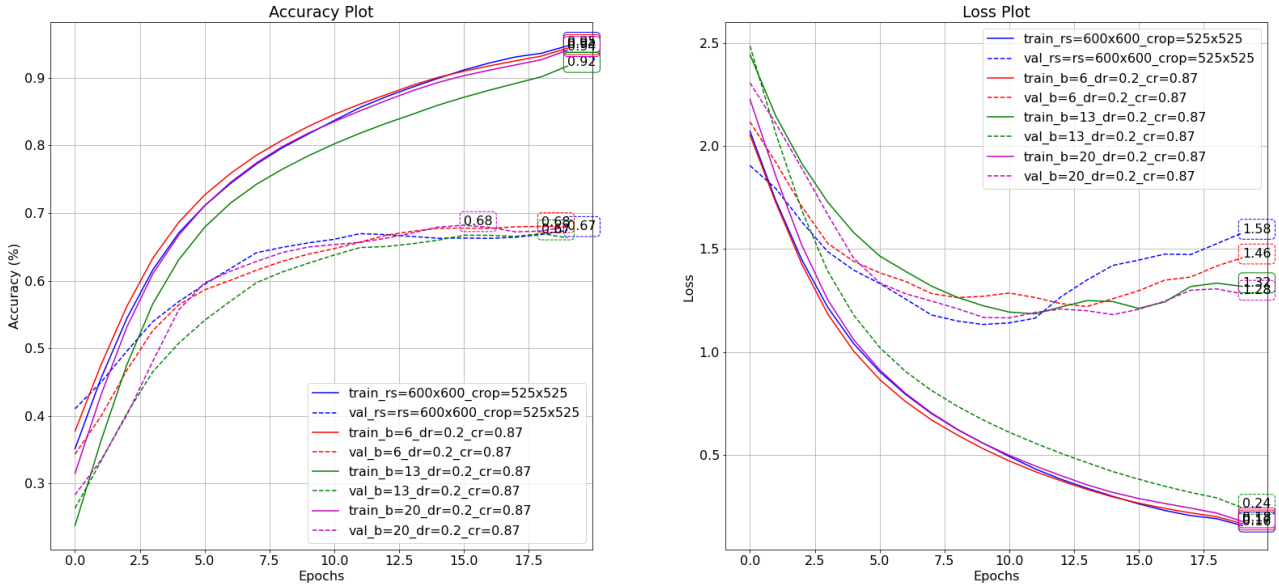


FIGURE 3.28: Padding experiment with 20% downratio- Bucketing: Accuracy & Loss over epochs

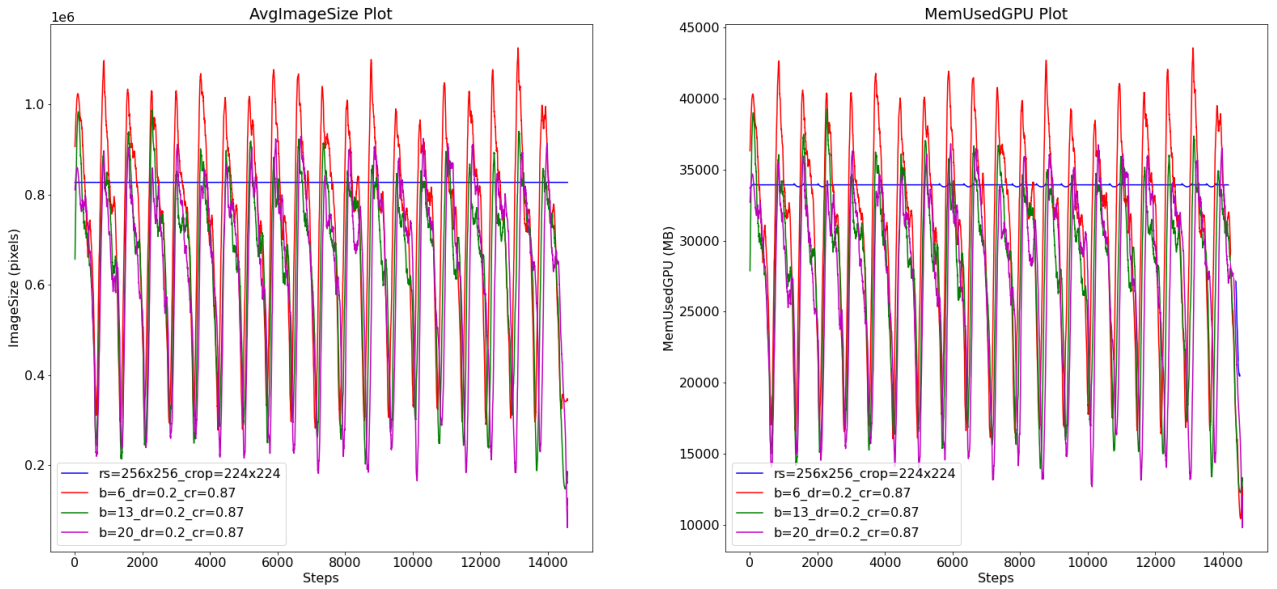


FIGURE 3.29: Padding experiment with downratio - Bucketing: Image size per batch & GPU Memory over epochs

Chapter 4

Conclusions

Throughout our work on *METavlitó* we wanted to find an approach to tackle challenges when we train variable-shaped images in high-resolution. Our proposed approach adopted some dynamic transformations and padding. Through this technique we made all images the same size maintaining their aspect ratio and then we put them together in the same batch. In order to overcome negative effects of extreme padding, when the sizes of the images differ a lot, we found a way for better padding minimization, through clustering.

To do that, we experimented with some different algorithms using classic and custom objective functions. We found that in average Padding Cost objective function eq.(2.8), proposed in this thesis, provides the lowest TPA and when we use it along with our custom *Random Padding Refinement* (RPR) clustering algorithm could surpass all the other combinations (*Figure 3.13*.

RPR clustering, seemed to be able to minimize better the TPA because of its cluster refinement procedure which can approach better a global solution given the initial starting points which are the initial buckets (like the initial centroids in KMeans). For this reason it takes more time than the other tested algorithms to finish, however, the total duration is less than two minutes for 23k images.

Using these results, we tested our initial hypothesis.

1. *Loss of information reduces CNN's performance* (Hypothesis 1)

   There is some evidence that larger images can give better accuracy and lower the loss value. In general, in extreme cases where we have very small or large images there is less overfitting (*Figure 3.18*, *3.19*) because it's harder for the network to extract patterns since on the one side we have less information and on the other hand there is much more information. This happens because of we used images in HR, so when we reduce a lot their size the CNN can still find distinguishable feature maps. However, providing bigger images we noticed an improvement in the accuracy between $2-4\%$ and 16% in loss. These improvements follow a pattern that may be useful for understanding the relationship between different input sizes and the accuracy as it can be seen in *Figure 4.1*.

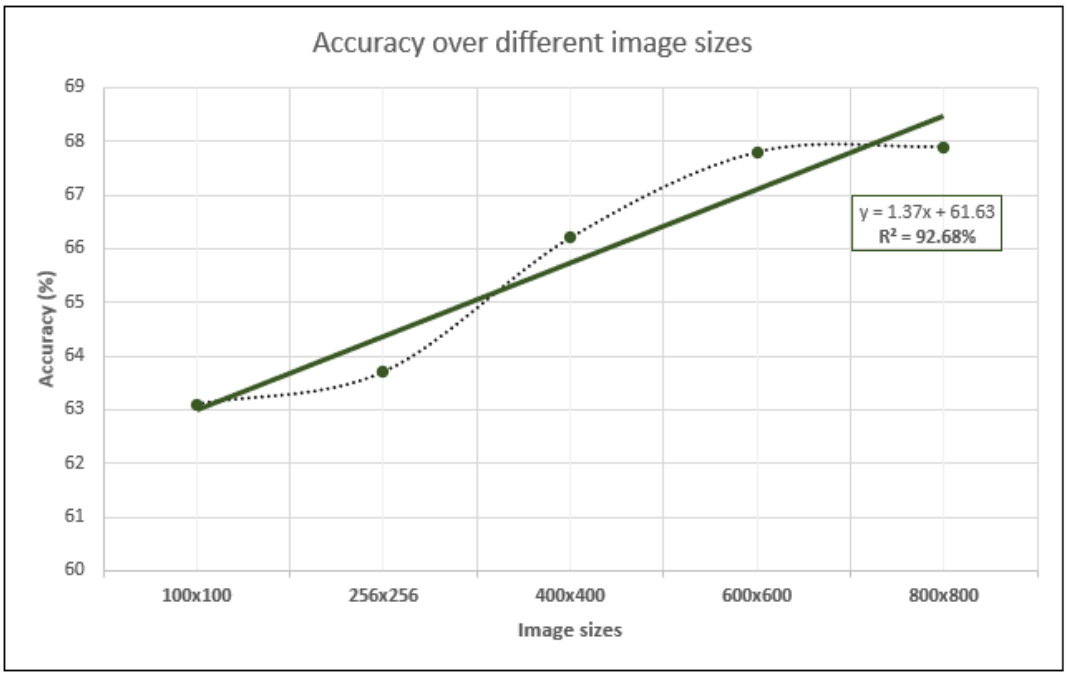


FIGURE 4.1: Linear Regression for the results of the experiments on Loss of Information

2. *Shape Deformation reduces CNN's performance* (Hypothesis 2)

   The results of this experiment were more unexpected. We ended up deforming extremely just one dimension leaving the other more or less untouched. This approach was necessary in order to test fairly the different models since all of them had very similar inputs in terms of number of pixels. The previous conclusion can be applied also here, where the network needs more time to learn larger images. A very interesting insight which came out of the experiments was that regularization capacity of deformation. The experiments showed similar results to data augmentation since we add prior knowledge (deformed images with the same labels) decreasing at the same time the variance of the model *Figure 3.21*.

   In more details, from deforming the images we managed around 4% of improvement in the accuracy and more than 43% in loss compared with the baseline (*Table 3.3*. However, we can not reject our hypothesis because the nature of the dataset included pictures in HR and very neutral background as can be seen for example in *Figure 3.1*. Thus, for better exploration of that topic is needed another dataset.

3. *Padding can increase CNN's performance with a slower convergence* (Hypothesis 3)

   When we use padding, limiting the maximum dimensions of the image to be the same with the ones of the baseline, we spotted 2% improvement in the accuracy with a very small overhead per epoch in time (see *Table 3.4*). Also, using bucketing we reached to approximately 3% of improvement in accuracy (see *Table 3.5*).

   Across the rest of the experiments we saw that the larger number of buckets the less time needed in each epoch during training and the less average memory needed. Furthermore, we could also notice that the more buckets we had

during training the less overfit we ended up having and in general terms we could see also some improvement in the accuracy (see *Table 3.7*; which again shows some regularization effects just from efficiently rearranging the batches validating our hypothesis.

In general, across these experiments we saw a better accuracy of 5% more than the baseline and 2% more than the model which takes the same input size. The loss is also improved lowering it by 37%. In different runs the accuracy results could fluctuate $\pm 1 - 2\%$. Finally, using our bucketing technique the GPU memory require in average $3 - 7$GB less memory than using pre-determined larger resized images accelerating the training time.

4. *Excessive padding can decrease CNN's performance* (Hypothesis 4)

   In the case of having really large images we saw that padding without bucketing (thus, with extreme padding) had really bad results, validating that extreme bucketing can actually decrease CNN's performance. When we set the maximum size of images even though it was giving similar accuracy and loss as the other models but it didn't improve the GPU nor the CPU memory, because most of the times because of the stochasticity in creating batches, it would include large images (see Table 3.4).

   We noticed the same when we resized the images with respect to the aspect ratio (with 10% and 20%) because there was an even bigger variability of image sizes with a lot of differences in their shapes and thus it required much more time for training and at the same time couldn't give good results (Table 3.6). For that reason we can validate our hypothesis and say that when we have many different shapes and scales of images the bucketing technique should be preferred more.

   In *Figure 4.2* we can see the relationship of the accuracy and the different number of buckets we tried in the experiments (note that $Buckets = 0$ corresponds to extreme padding). Also, we can compare the evolution of accuracy between setting a constant maximum image size and using dynamic resizing (down ratio) of 10% which implies the benefits of using larger images as we saw in the experiments of *Loss of information* above. With this we can get an idea of the effect of different number of buckets is each case.

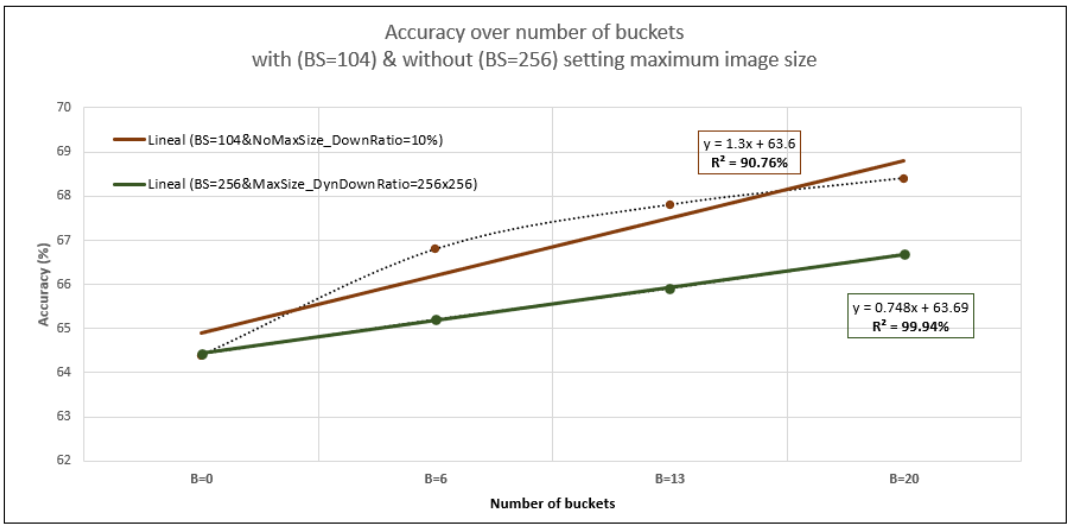


FIGURE 4.2: Linear Regression for the results of the experiments on Padding

In conclusion, the proposed *Random Padding Refinement* clustering algorithm combined with the *bucketing* method, that we introduced in this work, reduces the padding and avoids deformation better than all considered alternatives. The relevance of this contribution is highlighted by the validation of the hypothesis made and evaluated. Our proposed method can actually be used and replace other options that use large images as input in a CNN. Our approach can take into account variable-shaped images in high-resolution giving some improvements in the performance and at the same time, requires in average less memory. By forcing bucketing to have maximum image size the same one we would use as pre-determined one (i.e. 224x224), bucketing can also be used as a way to incorporate in the model the information of the different shape of images with the trade off of more time needed per epoch and more time to converge because of the bigger number of batches. At the same time, it can work as a regularization technique lowering the variance of the model.

4.1 Future work

As a future work of *METavlitó* we can include more sophisticated methods of bucketing which takes into account the size of the bucket. Also, something that could be interesting as a future extension of the *METavlitó* project is the dynamic assignment of the batch size based on the bucket or the image size of the batch changing appropriately the learning rate. Apart from that we need to do more experiments with different image processing tasks like regression, superresolution or object detection which may show bigger advantages of our approach.

Apart from that we need to do some more experimentation with some more sophisticated architectures in order to see also if this approach can indeed be applied globally no matter of the architecture. Finally, some more work is needed in resizing extreme cases like panoramas which can bring many troubles when we tried to resize them with either technique.

Chapter 5

Appendix

5.1 Clustering Essentials

Clustering can be used as a pre-processing mechanism to cluster together images and control the batches during the training or validation stage. The general goal of clustering is, given the number of clusters, to find meaningful groups of data points that share common characteristics and are more similar to the ones in the group than with the ones out of the group. This group can be called cluster or bucket. Each algorithm uses some kind of dissimilarity metric (usually the Euclidean distance/L2-norm) to distinguish not related groups and differs in the way it partitions the groups. In our use case the measure of similarity has to do with the images' shape.

5.1.1 KMeans based clustering

KMeans algorithm is an iterative algorithm that tries to partition the dataset into K predefined distinct non-overlapping subgroups (clusters) where each data point belongs to only one group. It tries to make the inter-cluster data points as similar as possible while also keeping the clusters as different (far) as possible. It assigns data points to a cluster such that the sum of the squared distance between the data points and the cluster's centroid (arithmetic mean of all the data points that belong to that cluster) is at the minimum. The less variation we have within clusters, the more homogeneous (similar) the data points are within the same cluster.

The way KMeans algorithm works is as follows:

- Specify number of clusters K.
- Initialize centroids by randomly selecting K data points for the centroids without replacement.
  - The version of KMeans++ picks just the first centroid randomly and the rest from a weighted probability distribution where a point x is chosen with probability proportional to the squared distance $D(x)^2$, in order to be as different as possible [3]
  - There is a version of Improved Canopy KMeans[38] (also developed in this work) where the centroids are pre-defined by an algorithm which is based on some density metrics.
- Compute the sum of the squared distance between data points and all centroids.
- Assign each data point to the closest cluster (centroid).

- Compute the centroids for the clusters by taking the average of the all data points that belong to each cluster.
  - The version of KMedoids the new centroids are the member of the cluster with the minimum sum of square distance to the rest of the cluster.
- Keep iterating until there is no change to the centroids. i.e assignment of data points to clusters isn't changing or is under a specified range of a tolerance value

The approach KMeans follows to solve the problem is called Expectation-Maximization. The E-step is assigning the data points to the closest cluster. The M-step is computing the centroid of each cluster. The objective function is the Euclidean distance:

$$\mathcal{J} = \sum_{j=1}^{n} \sum_{k=1}^{K} w_{jk} \cdot \left\| x_j - \mu_k \right\|^2 \tag{5.1}$$

where $w_{jk} = 1$ for data point x_j if it belongs to cluster k; otherwise, $w_{jk} = 0$. Also, μ_k is the centroid of xi's cluster.

The way to find the clusters and calculate each time the representatives of the clusters the centroids, is as follows for the given objective function which actually is the euclidean distance:

$$\frac{\partial \mathcal{J}}{\partial w_{jk}} = 0 \Rightarrow w_{jk} = \begin{cases} 1, & k = arg\min_l \left\| x_j - \mu_l \right\|^2 \\ 0, & \text{otherwise} \end{cases} \tag{5.2}$$

Concerning the M-step we want to find the right centroids:

$$\frac{\partial \mathcal{J}}{\partial \mu_k} = 0 \Rightarrow \mu_k = \frac{\sum_{j=1}^{n} w_{jk} \cdot x_{jk}}{\sum_{j=1}^{n} w_{jk}} \tag{5.3}$$

5.1.2 Hierarchical Clustering

In this project we also study different techniques of hierarchical clustering where the clusters are tree-organized which infers that each cluster consists subclusters so, in this type of clustering we have overalapping clusters, as can be seen in *Figure 5.1*. Each technique uses different dissimilarity metric to calculate the proximity between the new cluster and the original clusters and finally form another sub-cluster.

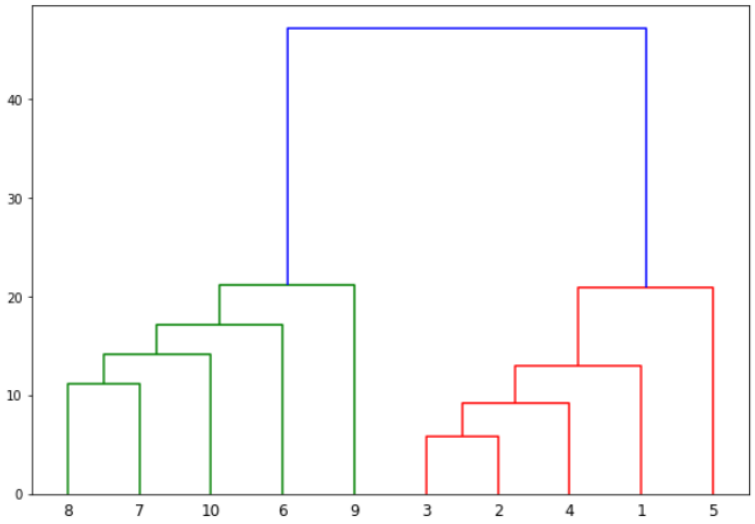


FIGURE 5.1: Hierarchical clustering example

Whereas KMeans tries to optimize a global goal (variance of the clusters) and achieves a local optimum, agglomerative hierarchical clustering aims at finding the best step at each cluster fusion (greedy algorithm) but resulting in a potentially sub-optimal solution. Agglomerative hierarchical clustering is easy to understand but it rarely provides good solutions and also is high in time complexity, generally, if the number of elements to be clustered is represented by n and the number of clusters is represented by K, then the time complexity of hierarchical algorithms is $O(n^2 logn)$.

Its general concept is to merge items into clusters based on a distance/similarity usually based on best pairwise similarity. Typically the steps are:

- each element is a cluster on its own
- compute similarity between all pairs of clusters and store the results in a similarity matrix
- merge two most similar clusters according to the linkage metric
  - Single (minimum distance between 2 pair of examples between two clusters)
  - Complete (maximum distance between 2 pair of examples between two clusters)
- update the similarity matrix
- repeat until everything belongs to the same cluster

Bibliography

[1] Jungmo Ahn et al. "Convolutional neural network-based classification system design with compressed wireless sensor network images". In: *PLOS ONE* 13 (May 2018), e0196251. DOI: 10.1371/journal.pone.0196251.

[2] Ali Alqudah et al. "Brain Tumor Classification Using Deep Learning Technique – A Comparison between Cropped, Uncropped, and Segmented Lesion Images with Different Sizes". In: (Jan. 2020).

[3] David Arthur and Sergei Vassilvitskii. "K-Means++: The Advantages of Careful Seeding". In: vol. 8. Jan. 2007, pp. 1027–1035. DOI: 10.1145/1283383.1283494.

[4] Simone Bianco et al. "Improving CNN-Based Texture Classification by Color Balancing". In: *Journal of Imaging* 3 (July 2017), p. 33. DOI: 10.3390/jimaging3030033.

[5] Francesco Bianconi et al. "Robust color texture features based on ranklets and discrete Fourier transform". In: *Journal of Electronic Imaging* 18 (Jan. 2009), pp. 043012–1. DOI: 10.1117/1.3273946.

[6] Léon Bottou. "Large-Scale Machine Learning with Stochastic Gradient Descent". In: *Proc. of COMPSTAT* (Jan. 2010). DOI: 10.1007/978-3-7908-2604-3_16.

[7] Wenlin Chen et al. "Compressing Neural Networks with the Hashing Trick". In: *Compressing Neural Networks with the Hashing Trick* (Apr. 2015).

[8] Xiaozhi Chen et al. "Multi-view 3D Object Detection Network for Autonomous Driving". In: July 2017, pp. 6526–6534. DOI: 10.1109/CVPR.2017.691.

[9] J. Deng et al. "ImageNet: A Large-Scale Hierarchical Image Database". In: *CVPR09*. 2009.

[10] Mukesh Jha (Team Name Auto Magic) Devin Shanahan Sonu Chauhan and Suresh Amrani (Team Neuro Knights) Roshan Adhikari Umar Farooq. "How we utilized gamma correction for increasing our training data ?" In: *Medium* (2017). URL: https://medium.com/giscle/how-we-utilized-gamma-correction-for-increasing-our-training-data-47c16a040adc.

[11] Jeff Donahue et al. "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition". In: *arXiv preprint* 32 (Oct. 2013).

[12] Jiansheng Dong et al. "An Efficient Semantic Segmentation Method using Pyramid ShuffleNet V2 with Vortex Pooling". In: Nov. 2019, pp. 1214–1220. DOI: 10.1109/ICTAI.2019.00-98.

[13] M. Everingham et al. "The Pascal Visual Object Classes Challenge: A Retrospective". In: *International Journal of Computer Vision* 111.1 (Jan. 2015), pp. 98–136.

[14] Clement Farabet et al. "Learning Hierarchical Features for Scene Labeling". In: *IEEE transactions on pattern analysis and machine intelligence* 35 (Aug. 2013), pp. 1915–1929. DOI: 10.1109/TPAMI.2012.231.

[15] Nicholas Frosst and Geoffrey Hinton. "Distilling a Neural Network Into a Soft Decision Tree". In: (Nov. 2017).

[16] Krzysztof Geras et al. "High-Resolution Breast Cancer Screening with Multi-View Deep Convolutional Neural Networks". In: (Mar. 2017).

[17] Swarnendu Ghosh, Nibaran Das, and Mita Nasipuri. "Reshaping Inputs for Convolutional Neural Networks -Some common and uncommon methods". In: *Pattern Recognition* 93 (Apr. 2019). DOI: 10.1016/j.patcog.2019.04.009.

[18] Ross Girshick et al. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation". In: *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*. CVPR '14. USA: IEEE Computer Society, 2014, pp. 580–587. ISBN: 9781479951185. DOI: 10.1109/CVPR.2014.81. URL: https://doi.org/10.1109/CVPR.2014.81.

[19] Kaiming He et al. "Mask R-CNN". In: (Mar. 2017).

[20] Kaiming He et al. "Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition". In: *Computer Vision – ECCV 2014*. Ed. by David Fleet et al. Springer International Publishing, 2014.

[21] Asifullah Khan et al. "A Survey of the Recent Architectures of Deep Convolutional Neural Networks". In: (Jan. 2019). DOI: 10.1007/s10462-020-09825-6.

[22] Alex Krizhevsky. "One weird trick for parallelizing convolutional neural networks". In: (Apr. 2014).

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105.

[24] Y. LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1 (1989), pp. 541–551.

[25] Tsung-Yi Lin et al. "Focal Loss for Dense Object Detection". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PP (July 2018), pp. 1–1. DOI: 10.1109/TPAMI.2018.2858826.

[26] William Lotter, Greg Sorensen, and David Cox. "A Multi-scale CNN and Curriculum Learning Strategy for Mammogram Classification". In: Sept. 2017, pp. 169–177. ISBN: 978-3-319-67557-2. DOI: 10.1007/978-3-319-67558-9_20.

[27] Shuang Ma, Jing Liu, and Chang-Wen Chen. "A-Lamp: Adaptive Layout-Aware Multi-Patch Deep Convolutional Neural Network for Photo Aesthetic Assessment". In: (Apr. 2017).

[28] Chollette Olisah and Lyndon Smith. "Understanding Unconventional Preprocessors in Deep Convolutional Neural Networks for Face Identification". In: (Mar. 2019).

[29] Suo Qiu. "Global Weighted Average Pooling Bridges Pixel-level Localization and Image-level Classification". In: (Sept. 2018).

[30] Ali Sharif Razavian et al. "CNN Features off-the-shelf: an Astounding Baseline for Recognition." In: *CoRR* abs/1403.6382 (2014). URL: http://dblp.uni-trier.de/db/journals/corr/corr1403.html#RazavianASC14.

[31] Pouyan Salavati and Hossein Mahvash Mohammadi. "Obstacle Detection Using GoogleNet". In: Oct. 2018, pp. 326–332. DOI: 10.1109/ICCKE.2018.8566315.

[32] Ryo Takahashi, Takashi Matsubara, and Kuniaki Uehara. "Data Augmentation using Random Image Cropping and Patching for Deep CNNs". In: *IEEE Transactions on Circuits and Systems for Video Technology* PP (Aug. 2019), pp. 1–1. DOI: 10.1109/TCSVT.2019.2935128.

[33] Jinzhuo Wang et al. "CSPS: An Adaptive Pooling Method for Image Classification". In: *IEEE Transactions on Multimedia* 18 (June 2016), pp. 1–1. DOI: 10.1109/TMM.2016.2544099.

[34] Jiaxiang Wu et al. "Quantized Convolutional Neural Networks for Mobile Devices". In: June 2016, pp. 4820–4828. DOI: 10.1109/CVPR.2016.521.

[35] Yunyang Xiong, Hyunwoo J. Kim, and Varsha Hedau. "ANTNets: Mobile Convolutional Neural Networks for Resource Efficient Image Classification". In: *CoRR* abs/1904.03775 (2019). arXiv: 1904.03775. URL: http://arxiv.org/abs/1904.03775.

[36] Yichong Xu et al. "Scale-Invariant Convolutional Neural Networks". In: (Nov. 2014).

[37] Matthew D. Zeiler and Rob Fergus. "Visualizing and Understanding Convolutional Networks." In: *CoRR* abs/1311.2901 (2013). URL: http://dblp.uni-trier.de/db/journals/corr/corr1311.html#ZeilerF13.

[38] Geng Zhang, Chengchang Zhang, and Huayu Zhang. "Improved K-means Algorithm Based on Density Canopy". In: *Knowledge-Based Systems* 145 (Apr. 2018). DOI: 10.1016/j.knosys.2018.01.031.