UNIVERSITAT DE BARCELONA

Treball final de grau

GRAU DENGINYERIA INFORMÀTICA

Facultat de Matemàtiques
Universitat de Barcelona

CAN A CNN RECOGNIZE MEDITERRANEAN DIET?

Author: Pedro Herruzo Sanchez

Supervisor:	Dra. Petia I. Radeva
Completed in:	Departament de Matemàtica Aplicada i Anàlisi

Barcelona, June 30, 2016

“A year spent in artificial intelligence is enough to make one believe in God.”

Alan Perlis

Abstract

Nowadays, we can find several diseases related with the unhealthy diet habits of the population, such as diabetes, obesity, anemia, bulimia and anorexia. In many cases, it is related with the food consumption of the people. Mediterranean diet is scientifically known as a healthy diet that helps to prevent those and other food problems. In particular, our work focuses on the recognition of Mediterranean food and dishes. It is part of a wider project that analyses the daily habits of users with wearable cameras, within the topic of Lifelogging. It appears as an objective tool for the analysis of the patient's behavior, allowing specialist to discover patterns and understand user's lifestyle to find unhealthy food patterns.

With the aim to automatic recognize a complete diet, we introduce a challenging multi-labeled dataset related to Mediterranean diet called FoodCAT. The first kind of labels contains 115 food classes with an average of 400 images per dish, and the second one is composed by 12 food categories with an average of 3800 pictures per class. This dataset will serve as a basis for the development of automatic diet tracking problems.

Deep learning and more specifically Convolutional Neural Networks (CNNs), are actually the technologies with the state-of-the-art recognizing food automatically. In our work, we adapt the best, so far, CNNs architectures for image classification, to our objective into the diet tracking. Recognizing food categories, we achieved the highest accuracies top-1 with 72.29%, and top-5 with 97.07%. In a complete diet tracking recognizing dishes from Mediterranean diet, enlarged with the Food-101 dataset, we achieve the highest accuracies top-1 with 68.07%, and top-5 with 89.53%, for a total of 115+101 food classes.

Acknowledgements

Almost a year has passed, and I have learned what a research career looks like. It is not just about reading papers and developing your own theories. It is more about discussing topics with your colleagues, listening to what the people may suggest, sharing your knowledge with others, and feeling excited finding something new.

I want to give my special thanks to Dra. Petia I. Radeva and to all the members of the CVUB research group, that is part of the Department of Mathematics and Computer Science of the University of Barcelona, for guiding me on this fantastic work with their experiences and knowledge. Special thanks to PhD Candidate Marc Bolaños for his great advices during this research.

I am enormously grateful to the university of Groningen by letting me use the Peregrine HPC cluster to do the computations. Special thanks to PhD Candidate Amirhosein Shantia, who allowed me to use a great computer at the Robot Lab whenever I needed it, enjoying the nice research atmosphere they have.

Thanks to Estefania Talavera, who showed me for the last months how a research life seems to be, and for sharing her motivation, that is driving my career into academia. Thanks for all the fun moments in Groningen.

I want express my sincere thanks to all my friends and specially to my fellow double-degree students, who have been joining me on this adventure for the last six years with plenty of good experiences.

Thanks to all new friends I met in Groningen for the amazing conversations we had, the fun times we shared, and the new perspectives of life they gave me.

Thanks to my family, especially my sweet mother Maricruz Sanchez, who teaches me how to love life, my brother Aitor Herruzo with whom I grew up with, and to my father Francisco Herruzo, who passed away when I was seventeen, giving me the responsibility to take care of my family and the motivation to do something good in this life. You will always be deep inside in our hearts.

Last but not least, I would like to thank you, Laia Llorens. For your constant support and help during all these years together, supporting me to realize my dream.

Contents

List of Figures

List of Tables

Abbreviations

TFG	**T**reball **F**inal **G**rau, Final Bachelor Project
NN	**N**eural **N**etwork
CNN	**C**onvolutional **N**eural **N**etwork
SR	**S**uper **R**esolution
AT1	**A**ccuracy **T**op-**1**
AT5	**A**ccuracy **T**op-**5**
NAT1	**N**ormalized **A**ccuracy **T**op-**1**
CM	**C**onfusion **M**atrix
FC	**F**ully **C**onnected

Symbols

η	learning rate
μ	momentum

Dedicated to all people who love science

Chapter 1

Introduction

Technology that helps track health and fitness is on the rise, in particular, automatic food recognition is a hot topic for both, research and industry. People around us have at least 2 devices, such as tablets, computers, or phones, which they use daily to take pictures. These pictures are commonly related to food; people upload dishes to social networks such as Instagram, Facebook, Foodspotting or Twiter. They do it for several reasons; to share a dinner with a friend, to keep track of a healthy diet or to show their own recipes. This amount of pictures are really attractive for companies, who are already putting so much effort to understand peoples diet, in order to offer personal food assistance and get huge benefits from that.

In research, the trend is applied differently. Obesity, diabetes, anemia, and other diseases, all are close related to food consumption. Many doctors tell patients to write a diary of their diet, trying to make them aware of what they are eating. Usually people do not care too much about that and also they think it is boring. Another approach that would work better, is to make the food diary by pictures with the phone, or even better, to take the pictures automatically with a wearable little camera. This last approach is part of a wider project that analyses the daily habits of users with wearable cameras, within the topic of Lifelogging. It appears as an objective tool for the analysis of the patient's behavior, allowing specialist to discover patterns and understand user's lifestyle to find unhealthy food patterns.

Looking at food habits, the Mediterranean diet is scientifically know as a healthy diet. For example, a growing number of scientific researches have been demonstrating that olive oil, operates a crucial role on the prevention of cardiovascular and tumoral diseases, being related with low mortality and morbidity in populations that tend to follow a Mediterranean diet [1].

1.1 Automatic food recognition

Deep learning and more specifically Convolutional Neural Networks (CNNs), are actually the technologies with the state-of-the-art recognizing food automatically. In this section, we present the best approaches in the field, in order to use it later.

First, let us introduce the two CNNs architectures that we will use in our work. The first one, *GoogleNet* [2], was responsible for setting the state of the art for classification and detection in the ImageNet Large-Scale Visual Recognition Challenge in 2014 *ILSVRC14* [3]. The second model is *VGG* [4], which secured the first and the second places also for the ImageNet *ILSVRC14* competition [3], in the localization and classification tasks respectively.

Second, let us introduce the first public database based on food: It is the *Food-101* dataset [5], containing 101 food categories, with 101'000 images, that we will use later. Many researchers have been working with this dataset achieving very good results on food recognition [6], or in both location and food recognition [7].

Other food related classification task that we are interested in, is to classify food categories. In our case, we will do it following a robust classification of Catalan diet proposed in the book "El Corpus del patrimoni culinari català" [8]. Other related work on that topic, such as classifying 85 food categories [9] or 50 food categories [10] are achieving good results.

1.2 Objectives

In this section, we list and describe the specific aims of this project.

1.2.1 Build a dataset including healthy food

As we show in subsection 1.1, the actual food datasets are built in order to achieve a good performance in the general challenge of recognizing pictures automatically.

Our goal is to present a method for food recognition of extended dataset based on Mediterranean food, as it is scientifically supported as a healthy diet. This dataset has been classified following two different approaches. On one side, the images has been classified based on the food dishes categories, and on the other side, in a more general food categories. As an example, our system will recognize a dish with chickpeas and spinach as the food class 'cigrons amb espinacs', but also as food category 'legumbres'.

1.2.2 Technological solution to recognize the dataset automatically

We claim to define robust approaches to recognize the datasets with the state of the art methods.

1.2.2.1 Convolutional Neural Networks to recognize Food dishes

First, we are interested in applying a Convolutional Neural Network to recognize the new healthy dataset, together to the dataset *Food-101* [5]. To achieve this food recognition challenge, we use pre-trained models over the large dataset ImageNet, such as GoogleNet [2] and the VGG [4], both, as we said before, winners in different categories of the *ILSVRC* competition [3]. Second, in order to recognize food categories, we compare the differences between fine-tuning a pre-trained model over all the layers, versus the same model trained only for the last fully-connected layer.

1.2.2.2 Techniques to improve the quality of the dataset and the recognition task: Super-resolution

It has been proven that large images resolutions improves recognition accuracy [11]. Therefore, we will base on a new method to increase the resolution of the images, based on a Convolutional Neural Network, known as Super-resolution. With that, our goal is to get a better performance in the image recognition task.

We will also explore different methods such as, balance all classes for both datasets, or reduce the resolution of *Food-101*, making it more similar to the resolution of our healthy dataset.

Finally, we will compare all methods and we will choose the one which gets a better performance.

1.3 Memory Structure

This document is organized in six chapters. The first one is the current introduction. In Chapter 2, we expose the background required to understand all other sections, we explain the used datasets, the procedure from a biological neuron until a Convolutional Neural Network, and the super-resolution method to increase images resolution. Chapter 3 explains all methods used to build the *FoodCAT* dataset, as well as the dataset description, for the two different labels; dishes and categories. In chapter 4, first we introduce the organization of the project, and then, we expose the general procedure to reproduce all the experiments, using a particular example just to be more clear. We end the chapter with the SR's implementation description. The results are depicted in Chapter 5 as follows; First, we introduce how to set up all experiments, second, we explain the theory of the evaluation metrics, third, the results for foods and categories recognition, and fourth, we give our personal opinion about a recognition system based con CNN. Finally, in the last chapter we expose the conclusions and the future work.

Chapter 2

Methodology

Image Classification problem is the task of assigning a label from a predefined set of categories to an input image. Instead of trying to create an algorithm to perform this task, we will take a data-driven approach. We will do the same strategy that we would do with a child: We are going to show to a CNN many samples for each category and then develop learning algorithms to learn from the appearance of each class.

In this section, first, we would specify from which dataset we are going to face the image recognition challenge. Later, we explain the needed background on Convolutional Neural Networks to understand all related concepts used. Then, we expose the specific models used to learn the features of our dataset, and finally, we explain how the method known as Super-Resolution works, increasing image resolutions in order to improve the image recognition.

2.1 Used datasets

To face the image recognition challenge over the Mediterranean Diet, we decide to create a dataset containing a subset of Mediterranean dishes. These representative classes are based on Catalan Food, and we explain how this dataset has been built in section 3.1. We call this dataset *foodCAT*, referencing to Catalan Food.

As *Food-101* dataset contains some classes that also belong to Mediterranean diet, e.g. "paella", and it also contains several non-related classes with the Catalan Food, we will use both datasets to implement our recognition system.

2.2 Convolutional Neural Networks

In this section, we expose the needed background about Convolutional Neural Networks, in order to understand the architecture and configuration of the used models, for the food image recognition task. With this knowledge, we will be able also to understand the different phases for the section implementation, and the different measures to evaluate the built classifier. First, we introduce the intuitive model for a simple neuron called *Perceptron* induced by a biological neuron, and we derive with that the notion of a Neural Network. Lastly, we introduce the CNN with all parameters that we use later.

All materials used in this section are token from the course *Neural Networks* attended at the University of Groningen, part of the Artificial Intelligence bachelor's program. The content of this course, at the same time, was created based on the book [12].

2.2.1 Biological neuron

Figure 2.1 shows a very much simplified diagram of a biological counterpart. However, from the artificial neural network point of view, a typical neuron, collects signals from others through a host of fine structures called dendrites. The neuron triggers electrical pulses called spikes, through a long and thin thread known as an axon, which splits into several branches. Each branch will join a dendrite of another neuron in a point of intersection called synapse, where the input electrical signal will be altered to inhibit or excite the activity in the connected neuron. Depending if a neuron receives enough excitatory input, it sends a spike of electrical activity down its axon. We say that a neuron is learning, when the effectiveness of the synapses is changed to influence differently to another neuron.

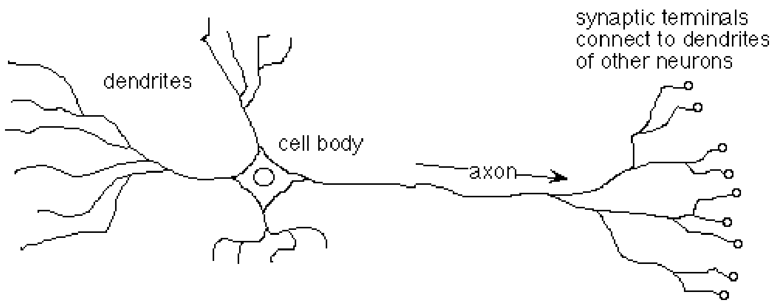

FIGURE 2.1: Simplified diagram of a biological neuron.

2.2.2 Perceptron

The Perceptron is an artificial neuron modeling the behavior of the biological neuron. Let us introduce few names to formalize this concept as an analogy of the biological.

Let x_i be the i-th input, as analogy of the i-th axon in the biological neuron. Let w_{ki} be the weight of the i-th input to the k-th neuron, as the synapse between the i-th axon and the i-th dendrite of the k-th neuron. Let b_k be the threshold of the k-th neuron, as the frontier that a biological neuron needs to beat, to be exited and generate an electrical spike.

With this concepts, we can define the *summing junction* v_k, as the sum of the weighted inputs, $v_k = \sum_i x_i w_{ki} - b_k$, and the *$k$-th output*, as a function of the summing junction, $y_k = \varphi(v_k)$. The function φ is called *activation function.* A basic example of the activation function could be the *step function* $\varphi \colon \mathbb{R} \to \{0,1\}$ defined by

$$\varphi(v_k) := \begin{cases} 1, & \text{if } v_k \geqslant 0, \\ 0, & \text{if } v_k < 0. \end{cases}$$

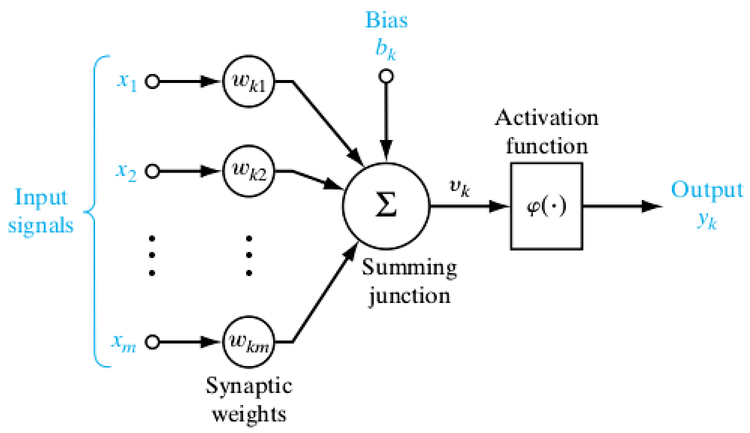

FIGURE 2.2: Perceptron, a model of the biological neuron.

In figure 2.2 we can visualize an artificial neuron with the specified notations. Same notations are useful to introduce a *Neural Network* (NN) or *Multilayer Perceptron*, which is nothing else than several Perceptrons connected, with an arbitrary structure, from n inputs $x_1, x_2, \dots, x_n$, to m outputs $y_1, y_2, \dots, y_m$. Tipically a NN is organized by layers; the input layer, the output layer, and the hidden layers, located between these two. The hidden and output layers are composed by Perceptrons, whereas the input layer are just values. Figure 2.3 shows a simple Neural Network topology.

2.2.3 Learning rule

In an artificial NN, learning means adaptation of the weights, as analogy of the change in the effectiveness of the synapses in a biological NN. Intuitively, we should keep learning until we perform the task without error, and that concept needs to be defined for this

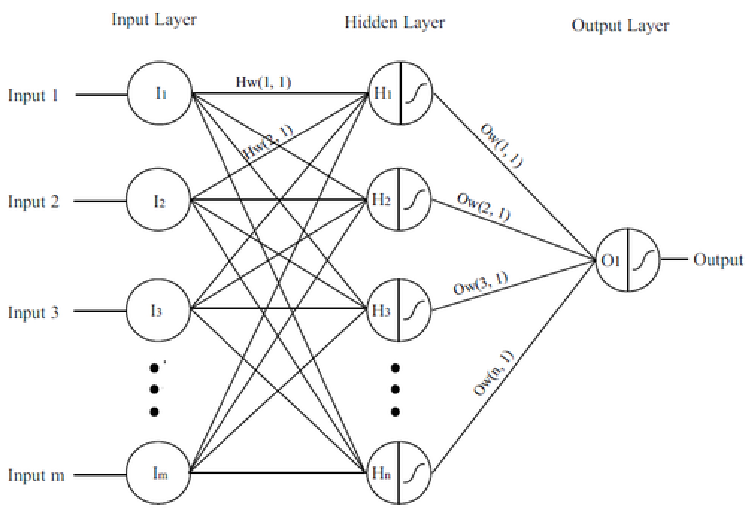

FIGURE 2.3: Artificial Neural Network composed by m inputs, n hidden neuron, and 1 output neuron.

context. In literature exists several different error functions, but we are going to use the *Euclidean*, just as an example to visualize the learning rule better.

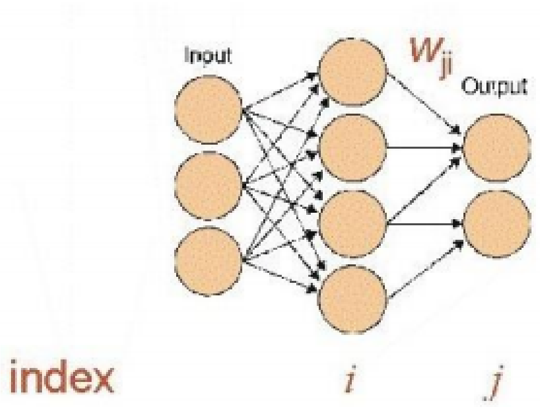

FIGURE 2.4: Artificial Neural Network with hidden layer i and output layer j.

Let us define these concepts from the architecture defined in figure 2.4, for an input pattern P:

- x_i^P, as the input values of the i-th neuron for the pattern P.
- t_j^P, as the target output of the j-th neuron for the pattern P.
- $v_j^P = \sum_i w_{ji} x_i^P - b_j$, as the activation of j-th neuron for the pattern P.
- $y_j^P = \varphi(v_j^P)$, as the predicted output of the j-th neuron for the pattern P.
- $e_j^P = t_j^P - y_j^P$, as the *error for the pattern* P in the single j-th neuron.

Then, we define the *error for the pattern P* as $e^P = \sum_j (e_j^P)^2$. The calculation of all outputs at the hidden and outputs neurons is known as *forward pass*, and the calculations of the errors in the outputs and hidden neurons is called *backward pass*. The strategy is to learn after each forward pass, calculating the errors, and uploading the weights of the NN with the goal to minimize the loss in the output neurons. It is called the *backpropagation* or *generalized delta rule*, using the gradient descent.[13]. We do not want learn indefinitely and we want to minimize the error globally rather than locally, there are few parameters to deal with this problems:

- The *learning rate* η sets the factor of the learning. The higher the learning rate is, the quicker the net will learn, if $\eta = 0$, the net will not learn.
- The *Momentum* μ is a trick to not fall in a local minimum. It is based in uploading the weights using the previous states of them.

Let us present the recipe to upload the weights following the scenario presented by figure 2.4, and then explain the pseudo-code of the backpropagation for a general architecture. For the following, a smooth activation function φ is required in order to apply the chain rule. With the same notations:

- $\frac{e^P}{\partial w_{ji}} = \frac{e^P}{\partial e_j^P} \frac{e_j^P}{\partial y_j^P} \frac{y_j^P}{\partial a_j^P} \frac{a_j^P}{\partial w_{ji}}$
- $\Delta w_{ji} = -\eta \frac{e^P}{\partial w_{ji}}$, or $\Delta w_{ji}(n) = -\eta \frac{e^P}{\partial w_{ji}}(n) + \mu \Delta w_{ji}(n-1)$, if we use the momentum.
- $w_{ji}\prime = w_{ji} + \Delta w_{ji}$

For a general architecture the backpropagation pseudo-code looks as follow:

```
while ( Error is larger than stop criterion)
        for (all training patterns)
                (forward pass)
                calulate the output of the hidden nodes
                calulate the output of the output nodes
                (backward pass)
                calculate the error in the output nodes
                calculate the error in the hidden nodes
                calculate the new weights
```

2.2.4 CNN

Convolutional Neural Networks are very similar to conventional Neural Networks, the main difference is that now the inputs are images, and therefore, we can apply certain properties allowing to reduce the amount of parameters making the execution of the net more efficient. The basic new concept that we need to feel comfortable before go further, is that in CNN, the layers have neurons arranged in 3 dimensions: width, height, and depth, e.g. in the case of the input layer as an image in RGB, the depth is 3 as the number of channels, and width and height are the spatial dimensions of the image. Let us say that the input images are 48x48x3. Then, each neuron in the input layer can be thought as a *3D volume* 1x1x3, and therefore, we have 48x48 3D neurons in the input layer. In order to understand the models in the next section, let us explain in detail what a convolution is, the common layers in a CNN, and the parameters that define the shapes of each layer.

2.2.4.1 Convolution

Let us show a simple example to understand the concept considering an image with size 6x6x3. Then, the filter or kernel that we want to apply to the image, must have the depth dimension equal to 3, to match the depth dimension of the input image. We have freedom to choose the width and height of the filter, while being smaller or equal to the original image, let us set it to 3x3. During the forward pass, we slide (more precisely, convolve) each filter across the width and height of the input image, starting with the two images aligned on the left top. In the first step, the entire filter will be align with the third first columns and rows of the input image. Then, we compute dot products between the entries of the filter and the input at any position for each channel and we add them, getting a single value. Next step we slide the kernel 3 positions to the right of the image. Therefore, the kernel is align with the first three rows and the last three columns. Again we apply the dot products and we get again a single value. Therefore, we have transformed the 6 columns and the first 3 rows of the input image in 2 values, i.e, from 3x6=18 cells to 2. Then, we apply again the filter with the half bottom of the image, and we obtain again 2 values. Now we have 4 values (a 2x2 square), when initially we had 6x6x3=108. Then, if we consider N filters instead of 1, and we apply the same mechanism, after apply this filters to the input image, we will have an output volume with dimensions 2x2xN. Figure 2.5 shows another example with a kernel 3x3 applied to one of the channels of an image 5x5.

2.2.4.2 Spatial arrangement

There are three hyperparameters controlling the shape of an output volume after apply a convolution or any of the other layers that we will explain later:

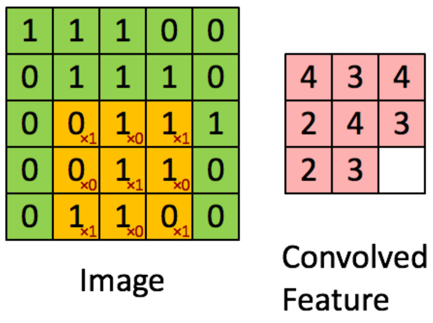

FIGURE 2.5: Convolution with 3x3 filter to one of the channels of an image 5x5. Source stanford.

1. *Receptive field (F)*: This is the size of the square filter we would like to apply. In our case F=3
2. *Depth (K)*: It corresponds to the number of filters we would like to use, each learning a different rule. In the example above, F=N.
3. *Stride (S)*: This is the number of pixels that we slide the filter. When the stride is 1, we slide the filters 1 pixel at time. In the example above, we use S=3.
4. *Zero-padding (P)*: Sometimes will be useful to pad the input volume with zeros around the border. The common use of that is to preserve the spatial size of the input volume, adding a zero-padding of 1 and using a filter 3x3, so we can apply to each pixel the convolution. In the above example we use P=0.

The formula to calculate how many spatial neurons the output volume has, is given by $(WF + 2P)/S + 1$. In the example above it is $(6 - 3 + 2 \times 0)/3 + 1 = 2$.

2.2.4.3 Layers

Now that we have the notions of 3D volume and convolution clear, let us list and briefly explain, the main layers that a CNN contains:

- *INPUT*: It Holds the pixel values of the image, normally the dimension are WxHx3, where W is width and H height.
- *CONV*: This are the convolutional filters explained in section 2.2.4.1. In general, a convolutional layer will have K filters of dimensions FxFxD, where D is the depth of the previous layer and F the Receptive field. If we are applying the convolutions to the images, D=3.

- *RELU*: This layer applies for each element of the previous layer an activation function, such as the *max(0,x)*. This does not change the dimensions of the input volume.
- *POOL*: This layer performs a downsampling operation along the spatial dimensions, changing the width and height, but not changing the depth.
- *FC*: Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks.

2.2.4.4 Caffe framework

There are several frameworks with high capabilities in the field of Deep Learning such as TensorFlow, Torch, Theano, Caffe, Neon, etc. We choose Caffe because the basis languages are C++ and Python, because it tracks the state-of-the-art in both code and models, and once you get used to it, it is fast for developing. We also decide to use it because it has a large community giving support on the Caffe-users group and Github, uploading new pre-trained models, that people like us, can use it again for different purposes. It allows a faster training and better accuracies [14] because normally these models are trained originally for larger datasets that the ones we will use on the fine-tuning.

2.3 Models

In this section we present the two models that we are using in our experiments. We explain why we choose them and we expose briefly their architecture. Both models are defined to take a image as an input with dimension 224x224x3, as a random crop of an image with resolution of 256x256x3.

2.3.1 GoogleNet

This deep convolutional neural network architecture is a replication of the model described in the GoogleNet publication [2]. It was the responsible for setting the state of the art for classification and detection in the ImageNet Large-Scale Visual Recognition Challenge in 2014 (ILSVRC14) [3]. The goal of this challenge was to recognize 1000 object categories and it was trained over 1.2 million images.

The files containing the net definition and the learned weights over the dataset (ILSVRC14), were downloaded from GitHub[1].

2.3.1.1 Topology

The network is 22 layers deep when counting only layers with parameters (or 27 layers if we also count pooling). Figure 2.6 shows the topology of the net, and make clear two of the features that made this net so powerful, as they explain in the paper [2]:

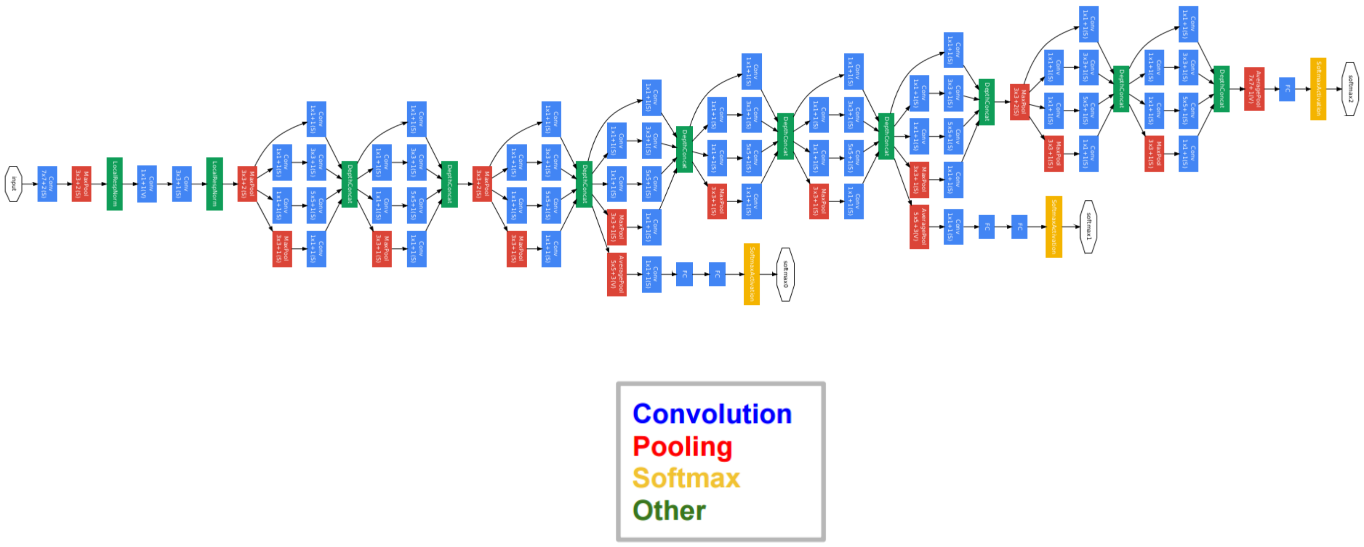

FIGURE 2.6: Topology of googleNet architecture containing 22-layer deep

1. Auxiliary classifiers connected to the intermediate layers (yellow layers in picture 2.6): This was thought to combat the vanishing gradient problem, given the relatively large depth of the network. During training, their loss gets added to the total loss of the network with a discount weigh. In practice, the auxiliary networks is relatively minor (around 0.5%) and it is required only one of them to achieve the same effect.

2. Inception modules: The main idea for it is that in images, correlations tend to be local. Therefore, in each of this 9 modules they use convolutions of dimension 1x1, 3x3, 5x5, and pooling layers of 3x3. Then, they put all outputs together as a concatenation. Figure 2.7 shows the structure of the Inception, and figure 2.8 shows the concrete pipeline. Note that to reduce the depth of the volume, convolutions 3x3 and 5x5 are performed after apply a 1x1 convolution, and pooling 3x3 is also

[1]https://github.com/BVLC/caffe/tree/master/models/bvlc_googlenet

followed by a convolution 1x1. This arguments makes the model more efficient reducing the number of parameters in the net.

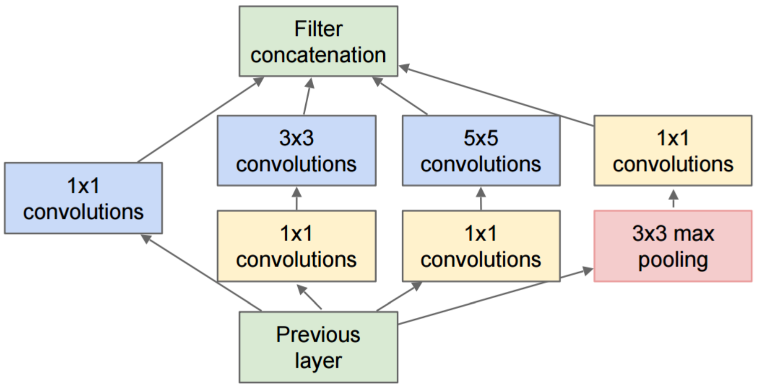

FIGURE 2.7: Topology of the inception module.

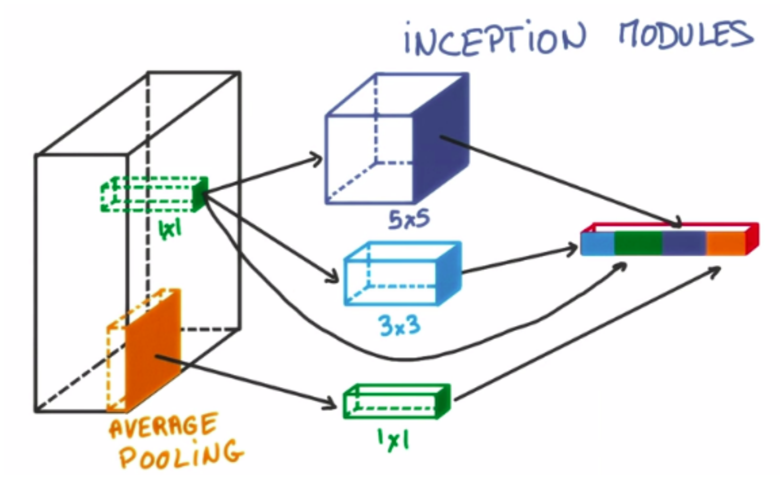

FIGURE 2.8: Topology of the inception modules visually described. The image is taken from a descriptive video in Udacity.

2.3.2 VGG-19

VGG model [4] secured the first and the second places for the ImageNet ILSVRC-2014 competition [3], in the localization and classification tasks respectively.

The learned weights over the dataset ILSVRC14, were downloaded from the Official website[2]. On another hand, their link to GitHub for downloading the net definition, and in this case, the net provided was deprecated for the new version of the framework Caffe. We changed all needed parameters in order to make it work, and we made it public (See appendix A for more details).

[2]http://www.robots.ox.ac.uk/~vgg/research/very_deep/

2.3.2.1 Topology

This net has 5 blocks of different depth convolutions (64, 128, 256, 512, and 512 consecutively) and 3 FC layers. The first 2 blocks contains 2 different convolutions each and the last 5 contains 4 different convolutions each. It is a total of $2 \times 2 + 3 \times 4 + 3 = 19$ layers. All convolutions have a kernel size of 3x3 with a padding of 1 pixel, i.e. the spatial resolution is preserved after convolution. Finally, after each convolutional block a max pooling is performed over a 2x2 pixel window, with stride 2, i.e. reducing by a factor of 2 the spatial size after each block. Section 2.3 in VGG-19 paper [4], confirm that small-size convolution filters are the key, together with apply **deep** CNN, to outperform googleNet in ILSVRC14 [3] in terms of the single-network classification accuracy. Figure 2.9 shows the VGG-19 structure explained above.

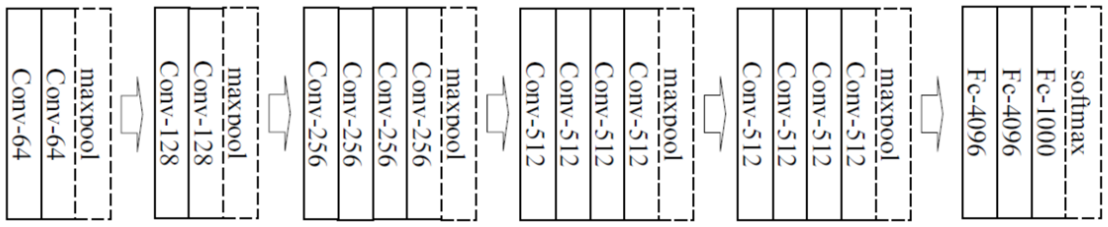

FIGURE 2.9: Topology of VGG-19 layers.

2.4 Super-resolution

The image dimensions of *FoodCAT* dataset are in average smaller than 256x256. Therefore, the images are increased to this size as an input for Caffe, causing deformation noise. We found interesting the fact that larger images improves recognition accuracy as it is show in [11]. In this paper, they demonstrate, there are many cases where the object size is small, and downsizing simply loses too much information. Motivated with that, we claim that increasing the resolution with a state-of-the-art method instead of leave the net do it with the regular resize, the image recognition accuracy is better.

To increase the size of the images we use the new method called super-resolution [15]. In this paper they show how obtaining a high-resolution (HR) image from a low-resolution (LR) can be perform by a combination of the conventional sparse coding model and CNN, achieving notable improvements over the generic CNN model in terms of both recovery accuracy and human perception.

2.4.1 Examples of SR for images in the used datasets

In this section we show the accuracy from a human perception, of the SR applied to two images, one from *Food-101* and the other from *FoodCAT*. The goal for these examples is to show that increasing the *FoodCAT* images with SR, to a size bigger or equal than 256x256, and then let Caffe decrease the resolution to 256x256, the resulting image looks better from a human perspective, than increasing the resolution from the original size to 256x256 by the standard resize, as Caffe does.

2.4.1.1 Example with Food-101

First we will show the behaviour explained with a image of resolution higher than 256x256. The goal is to see that the original reduced to this size, looks very similar to the same image decreasing the resolution to smaller than 256x256, and then applying the procedure explained before for the *FoodCAT* images.

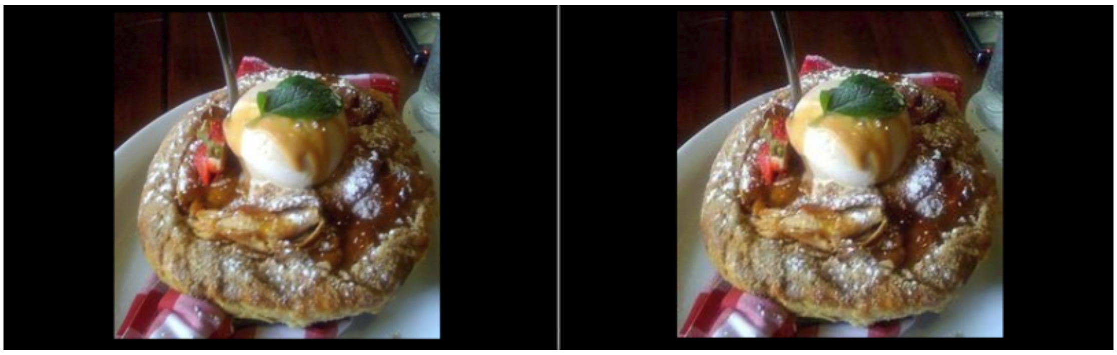

FIGURE 2.10: Example of SR used in a hight resolution image. Left: original image 512x512 resized to 256x256. Right: Original reduced at 40% 230x230, then increased by SR x2 to 460x460, and finally resized to 256x256.

With our eyes we can not see any difference in figure 2.10, both images looks very similar. Also, computing standard image measure, the difference in not even big, as it is shown in table 2.1. We also show the histogram for the two images in figure 2.11. We conclude, from a human perspective, than the method is working good for this example.

	mean	max	min
Original	91.10	255	91.10
SR	91.04	255	91.04

TABLE 2.1: Mean, max, and min computed for two images. Original: original image 512x512 resized to 256x256. SR: Original reduced at 40% 230x230, then increased by SR x2 to 460x460, and finally resized to 256x256.

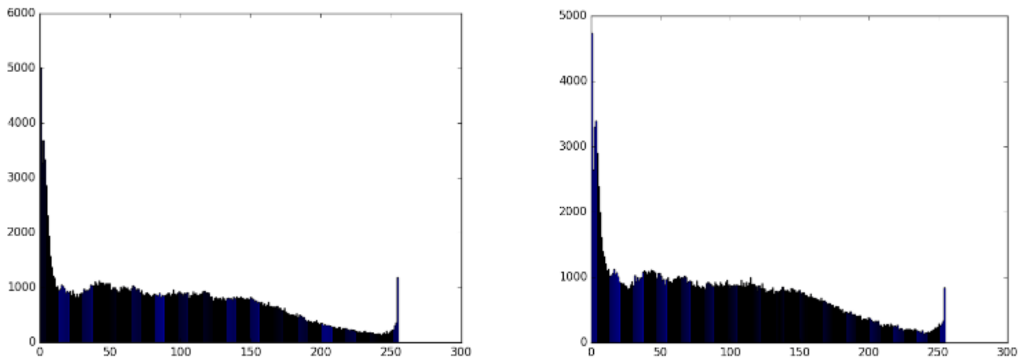

FIGURE 2.11: Distribution of pixels amongst those grayscale values. Left image is the original, and right with the SR.

2.4.1.2 Example with FoodCAT

In this example we choose a random image from FoodCAT dataset, and we applied SR in order to get that both image dimensions, width and height, to be bigger or equal than 256. Original image is 402x125, so the SR was applied with a factor of 3, because 2 was not enough ($2 \times 125 < 255$). Figure 2.12 shows how the image with SR looks much more better than the original, when they are resized to 256, from a human perspective.

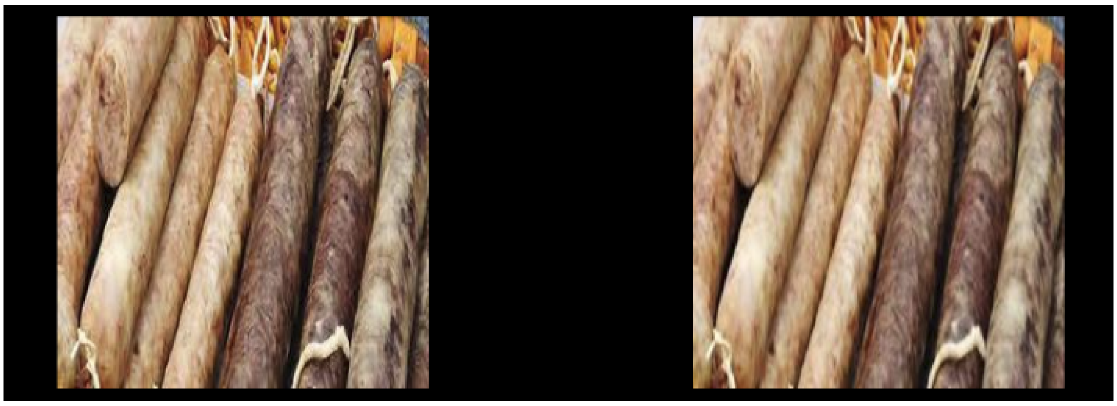

FIGURE 2.12: Left shows the SR decreased to 256x256 and right shows the original increased to 256x256.

Chapter 3

Construction of the healthy food dataset

In this section we show how we created the dataset *FoodCAT* with two different labels: Catalan dishes, and Catalan food Categories. We will refer as *FoodCAT* when we speak about the first kinf of labels, and as *FoodCategories* for the second. This creation involve: find all Catalan food dishes, select a sample, get images for each class and clean the non-related images.

3.1 FoodCAT

Mediterranean Diet is based on several dishes that involves many countries along the Mediterranean Sea. We choose the Catalan cuisine as a representation of this diet because we are from Catalonia, so it is easy to choose a representative sample of the Catalan food. This is the reason why we call it *FoodCAT*, where *CAT* is used refer to Catalonia.

3.1.1 Data scraping

First step we made is generate a database with a big list of the greatest and more representative names of Catalan cuisine. For that purpose we used "El Corpus del patrimoni culinari català" [8], which is the longest database of Catalan recipes that exists so far. This book also provides a wonderful work classifying each of this dishes among 12 food categories.

The website cuinacatalana[1] provides a large subset of these recipes. We created the script `tools/scripts/scrapCuinaCatalana.py` to download all these dishes on the file `tools/raw_data/food.json`, with the fields ”category”, ”name” and ”ingredients”. Here is an example of a food dish downloaded from this website:

```
{"category": "Caracoles", "name": "Bunyols de cargols", "ingredients": ["1 kg
  de cargols grans.", "150 g de farina.", "20 cl de llet.", "1 ceba.", "4 alls.
  ", "Una mica de llevat en pols.", "Julivert.", "Oli.", "Sal.", "Aigua."]}
```

A total of 904 dishes names can be found in the file specified above, and almost all dishes belong to 1 of the 12 food categories. From this big collection we selected manually the 140 more popular dishes, always trying to respect the percentages of dishes by categories. We show in table 3.1 these values, where the first column lists the categories, the second and third show the original dataset, and the fourth and fifth the selected dataset. A funny anecdote watching the percentages from the original dataset and the selected, is that we can say how much we like desserts.

	# dishes	%	# selected dishes	%
Carnes	223	24,67	26	18,57
Pescados y mariscos	156	17,26	25	17,85
Postres y dulces	123	13,61	34	24,28
Pastas, arroces y otros cereales	91	10,07	11	7,85
Verduras y otras hortalizas	79	8,74	11	7,85
Sopas caldos y cremas	71	7,85	8	5,71
Huevos	46	5,09	5	3,57
Ensaladas y platos frios	34	3,76	5	3,57
Caracoles	23	2,54	3	2,14
Legumbres	23	2,54	6	4,28
Salsas	20	2,21	4	2,85
Not available	11	1,22	0	0
Setas	4	0,44	2	1,42
Total	904	100	140	100

TABLE 3.1: Number of dishes per category. First column lists the categories, the second and third column show the original dataset, and the fourth and fifth the selected dataset.

3.1.2 Image scraping

Let us explain how we obtained the images related to the dataset of the selected food dishes introduced in the previous section. We created the script located in the path `tools/scripts/imageRetrieve.py` to download and save a sample of 1000 pictures related to a query using Google images. The other script created to find the images is located at `tools/scripts/findImages.py`, and it reads the file `tools/raw_data/`

[1] http://www.cuinacatalana.eu/ca/pag/receptes/

`selectedFoods.json`, where we saved all selected food classes. Then, one bye one, we used `tools/scripts/imageRetrieve.py` to download the images related to the corresponding class.

3.1.3 Image cleaning

Almost 1000 images were downloaded for each of the 140 selected food classes, but, just an average of 400 images were related with the corresponding class. We had been cleaning manually all these non-related images for each dish. Once we had chosen the best images, we decided to use in our dataset just the classes with at least 100 images. Therefore, the number of classes decreased from 140 until 115, and the final summary is a dataset with **115 dishes** with an average of **400 images per class**.

3.1.4 Final datasets

As we introduced in section 2.1, we are using the two datasets *FoodCAT* and *Food-101* for the food-recognition task. Let us first summarize in table 3.2 the number of images used for the complete datasets and the balanced one, that we explain later in detail.

	training	validation	testing	total
Complete	116.248 (80.800+35.448)	14.540 (10.100+4.440)	14.516 (10.100+4.416)	145.304(101.000+44.304)
Balanced	73.085 (40.400+32.685)	9.143 (5.050+4.093)	9.124 (5.050+4.074)	91.352(50.500+40.852)

Table 3.2: Number of images per phase (training, validation and testing) over the complete dataset and the balanced one. The values are presented giving the total first, and then, inside the brackets, giving first for Food-101 and then for FoodCAT.

All pictures in *Food-101* dataset have one dimension (width or height) equal to 512, whereas that pictures in *FoodCAT* does not follow any pattern, and in average, the resolution is below 256x256. In addition, the classes joining the two datasets are far to be balanced. This is the reason why we presented a combinations of both datasets changing the resolutions or balancing the classes, because we want to study how the model's learning process is influenced by these. Below, we present three different combinations changing resolutions, each of them presented in two forms; with the original classes and the balanced classes.

3.1.4.1 Original datasets

Here we present two datasets; 'foodCAT_OLD', which is just the union of *FoodCAT* and *Food-101*, and 'foodCAT_500', consisting on the dataset foodCAT_OLD, but taking, at most, 500 images per class.

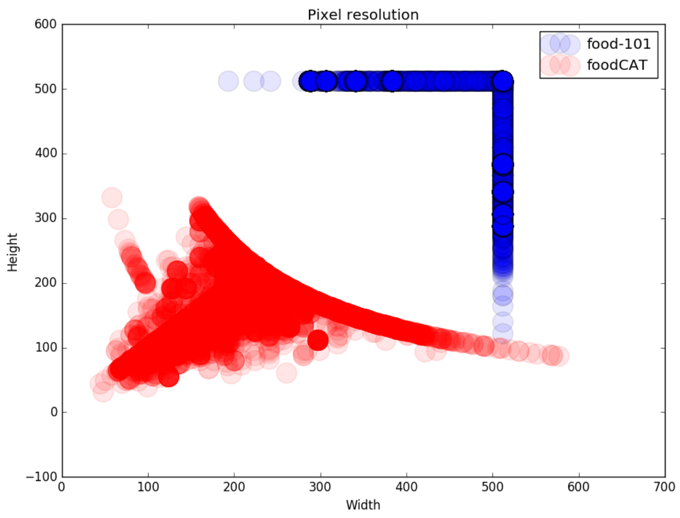

Figure 3.1: Images resolutions of FoodCAT and Food-101.

Figure 3.1 shows the dimension of all images for each dataset in foodCAT_OLD. We can observe how *Food-101* dataset follows the pattern to have one dimension (width or height) equal to 512, and *FoodCAT* has a huge diversity of resolutions, but in average, lower than 256x256.

In section 2.3 we explain that the used models transforms the resolution of the input images to 256x256. Therefore, images of *FoodCAT* are increased and deformed to 256x256, whereas that images of *Food-101* are decreased and deformed for the training. Thus, we are using images with much more noise in *FoodCAT* than in *Food-101*. The next two different datasets are created in order to face this problem.

3.1.4.2 Food-101 with resolution halved

In this dataset we decrease the resolution of all images in *Food-101*, to force the network to increase the resolution, and therefore add some noise to the images, as it is done with *FoodCAT*. The resulting datasets are named 'foodCAT_resized', which is just the union of *FoodCAT* and *Food-101* resized, and foodCAT_resized_balanced, consists on the dataset 'foodCAT_resized', but taking at most 500 images per class. Figure 3.2 shows the resulting resolutions for the datasets.

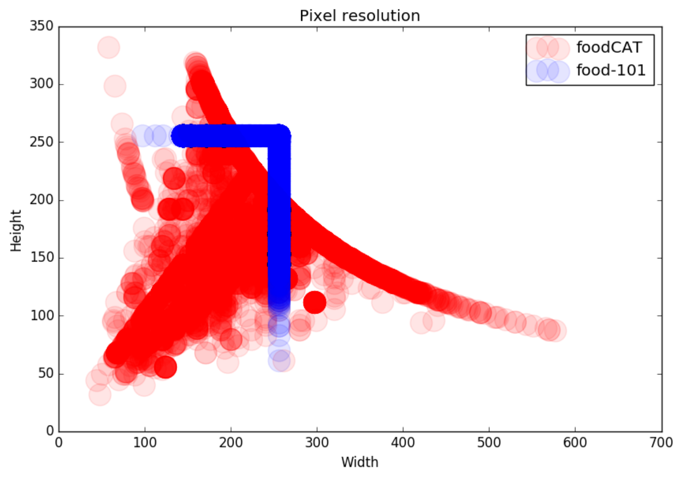

FIGURE 3.2: Images resolutions of FoodCAT, and Food-101 with resolution halved.

3.1.4.3 FoodCAT with resolution increased with super-resolution

In this dataset we increase the resolution of all images in *FoodCAT* with the SR technique, to later force the network to decrease resolution, and therefore have less noise in the images, trying to copy the behavior that is done with *FoodCAT*. The resulting dataset named 'foodCAT_SR' is the union of *FoodCAT* resized by SR and *Food-101*. The resulting dataset named 'foodCAT_SR_balanced', consist on the dataset foodCAT_SR, but taking at most 500 images per class. Figure 3.3 shows the resulting resolutions for the datasets.

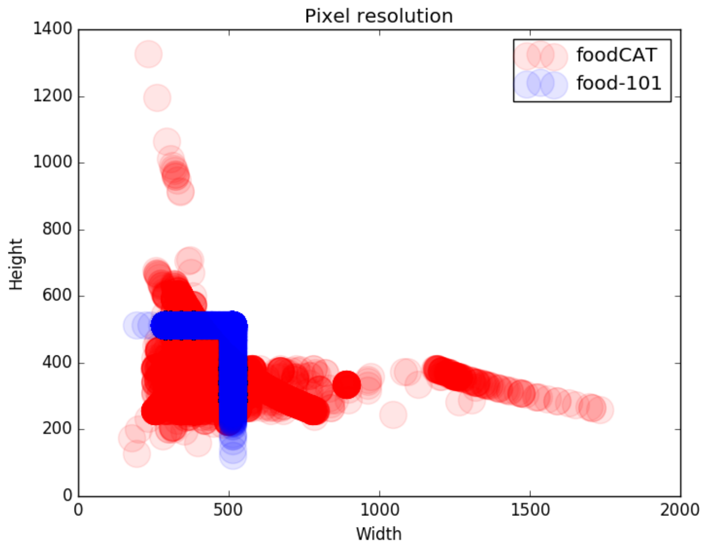

FIGURE 3.3: Images resolutions of FoodCAT with resolution increased by SR, and Food-101.

3.2 FoodCAT spplited by categories

To create this dataset we used the images from *FoodCAT* and the information about which category they belong, according to the website cuinacatalana[2], that is based in the book ”El Corpus del patrimoni culinari català” [8]. Table 3.3 shows the number of images per category.

	# Images
Postres y dulces	11.933
Carnes	7.373
Pescados y mariscos	5.977
Pastas, arroces y otros cereales	4.728
Verduras y otras hortalizas	3.007
Ensaladas y platos frios	2.933
Sopas caldos y cremas	2.857
Salsas	2.462
Legumbres	1.920
Huevos	615
Caracoles	470
Setas	438
Total	44.713

TABLE 3.3: Number of images per category.

[2] http://www.cuinacatalana.eu/ca/pag/receptes/

Chapter 4

Implementation

In this chapter we expose a step by step guide of our implementation. We would like, first, to let you know how the project is organized and which tools we use. Second, how to reproduce our experiments using the framework Caffe , including our new approaches to some lacks of this tool. And third, we explain the implementation of the Super-resolution method, that is very important in order to apply the CNN on a dataset of images with different resolutions.

4.1 Project Organization

Let us explain briefly how we organized the project, which tools we used, and show the basic pipeline. We used a local machine to built the dataset and to test the models. On another hand, all training of the models was done on a cluster of computers.

4.1.1 Local Machine

The local computer used was a a Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz. Here we set up all experiments that after we execute on the cluster.

4.1.2 Cluster

All CNNs trainings were performed on the Peregrine HPC cluster[1], provided by the University of Groningen. This cluster contains many nodes (computers) with different features, but we use just single nodes with the next following components:

[1]`https://redmine.hpc.rug.nl/redmine/projects/peregrine/wiki`

- 24 cores @ 2.5 GHz (two Intel Xeon E5 2680v3 CPUs)
- 128 GB memory
- 1 TB internal disk space
- 2 Nvidia K40 GPU accelerator cards

All instructions used to connect with the cluster, synchronize the cluster with the local computer, submitting jobs on the cluster, and another useful commands, can be found fully explained in the file `/utils/docs/peregrineCLUSTER`. Let us show the two instructions we use more often to synchronize both computers. We execute these instructions from the local machine, so, the instruction 'push', sends all new information from the local machine to the cluster and the instruction 'pull' does it in the other way around. Both instructions look as follow, respectively:

```
rsync -aPv TFG/ s3021610@peregrine.hpc.rug.nl:/data/s3021610/TFG/
rsync -aPv s3021610@peregrine.hpc.rug.nl:/data/s3021610/TFG/ /home/pedro/TFG/
```

In the file mentioned we can find plenty of flags that allow us to, for example, delete on the cluster what we delete on the local machine because the default 'push' is deleting nothing.

4.1.3 GIT

All projects longer than one month should use a control version tool. In our case, it was not used to recover ever a previous snapshot of the project. Instead of that, we use the portability that allows us to combine this tool with a remote repository, in our case GitHub. We want to remark how useful it is describe nicely each commit we do, so every time we start to work, we can have a brief summary explaining what we did on the previous commits.

4.1.4 Project Pipeline

We basically use GitHub as a remote repository to save all created files, avoiding the images and the files .caffemodels and .solverstate that Caffe generates, because they are too heavy, and GitHub has a limited memory for the free version. The training of the models is performed by the cluster, which we maintain synchronized with the local computer, including the files that we avoid with GitHub.

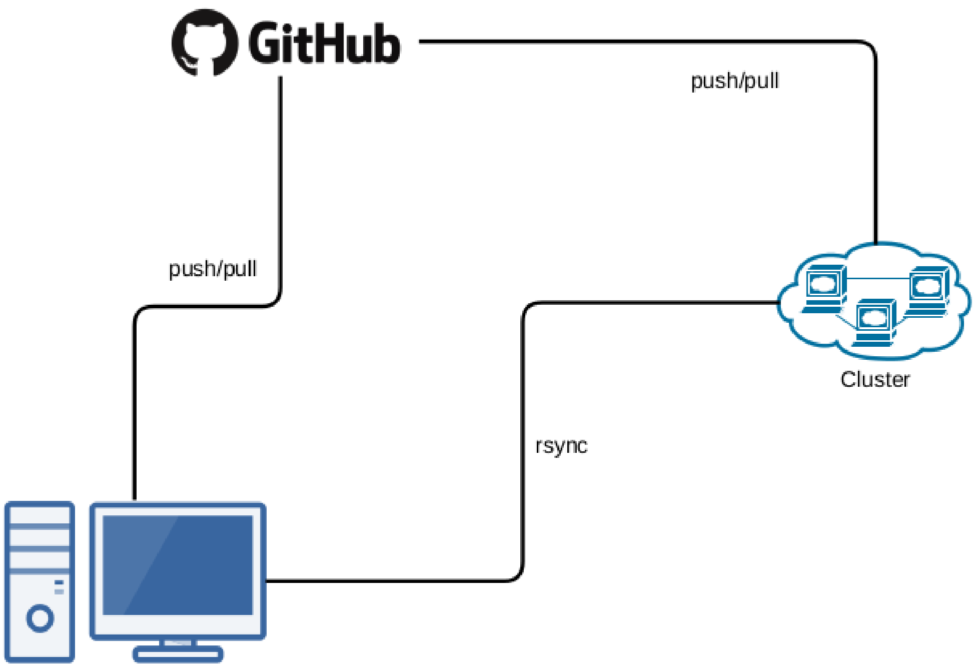

Figure 4.1: Project pipeline. GitHub is used as a remote repository and the cluster is used in the training of the CNN models

4.2 Recognizing Catalan Food

In this section we expose all required steps in order to reproduce any of our experiments. We start listing the requirements for the framework that we are using, then we explain in detail how to set up a model to train, next we describe the training, and last we introduce new methods to choose and test the best learned model. We end the section giving some tips about how to make the project portable to work with Caffe in any computer.

We avoid briefly the installation instructions because other people already did successfully, so we just provide the link[2] with the instructions we used.

4.2.1 Caffe Requirements

To fine-tuning a model it is needed:

1. *dataset*: We use both datasets *Food-101* and *FoodCAT*

[2]`https://github.com/tiangolo/caffe/blob/ubuntu-tutorial-b/docs/install_apt2.md`

2. *train, validation and testing*: .txt files linking the images with the class number they belong to.

3. *model*: .prototxt file defining the net architecture. There we also need to set:

    (a) On data layers TRAIN and TEST, we need to specify the path of the train and validation .txt files respectively.

    (b) On the last inner product layer, we define the number of classes that the net will be able to distinguish.

    (c) Note that we can set η to zero to an arbitrary layer. It will mean that we do not want to modify the learned weights from the loaded model on that layer.

4. *solver*: .prototxt file defining the net parameters and the model file.

5. *snapshot*: .caffemodel file with the weights we want to tune, that must match with the model definition.

4.2.2 Caffe Structure

Caffe is a really flexible tool that mainly allows us to work with the folders and files structure that we desire. Here we introduce how we do it, trying always to be easy and useful. All our scripts are made in order to follow that structure, so, to reproduce our implementation without changing the internal code, this structure is required.

The structure depicted below requires the items listed in section 4.2.1. We are not defining all folders and sub-folders that our project contains in this section, but we will introduce and explain the most relevant ones. When we require scripts in folders that are not described here, we give their relative path from our main folder.

Our main folder is called `TFG`, and it is structured as follows:

```
TFG
├─ data
│  ├─ food-101
│  │  └─ images
│  └─ foodCAT-db
├─ foodCAT
│  ├─ classesID.txt
│  ├─ train.txt
│  ├─ val.txt
│  ├─ test.txt
│  └─ test_just_foodCAT.txt
└─ models
   └─ googlenet_SR
      ├─ train_val.prototxt
      ├─ solver.prototxt
      ├─ test.prototxt
      ├─ test_just_foodCAT.prototxt
      ├─ LOGS
      └─ snapshots
```

- *data*: Inside the folder data, we have two different datasets, Food-101 and FoodCAT-db. Each dataset contains n and m class respectively, where each class contains N_i images with $1 <= i <= n$ for food-101 dataset, and M_j images with $1 <= j <= m$ for foodCAT-db.

- *foodCAT*: Contains the corresponding class for each image, divided in the training, validation and test sets. An example about how each line in this files looks is shown in section 4.2.8. The file classesID.txt" contains the correspondence between a class and the associated value.

  This folder with all files, except the "test_just_foodCAT.txt" file, is automatically generated given the path to the datasets we want to use. Our script to do it, is called as follow:

  ```
  ~/TFG/buildNET/builder$ python  buildNET.py -p '../../data/foodCAT-db
    ','../../data/food-101/images' -t '../..'
  ```

The file "test_just_foodCAT.txt" is a copy of the file "test.txt", but deleting all lines with pictures from the food-101 dataset.

- *models*: For each different model, we create a folder with the structure as it is shown for the branch "googlenet_SR" in the tree in section 4.2.2. The file "train_val.prototxt" is the model explained in section 4.2.1. The file "test.prototxt" is used in section 4.2.6 to test the performance of the learned model. It is a copy of "train_val.prototxt" deleting the TRAIN layer and linking the TEST layer to the path `TFG/foodCAT/test.txt` instead of to `TFG/foodCAT/val.txt`. The file "test_just_foodCAT.prototxt" is a copy of "test.prototxt" pointing the TEST layer to `TFG/foodCAT/testJustFoodCAT.txt`.

  In "LOGS" we save our training output, pictures and results of the current model. In the folder called "snapshots", is where the net saves the files .caffemodel (weights at certain iteration) and .solverstate (all parameters and weights at certain iteration). We define where to save the snapshots in the file "solver.prototxt", where we also define the net parameters and the model.

All interesting scripts, tips or instructions not explicitly mentioned on this document can be found in the path `TFG/tools/scripts` or `TFG/utils/docs`, respectively.

4.2.3 Net Parameters

In Caffe , the file "solver.prototxt" has the following responsibilities:

1. To create the training network for learning and the validation network for testing
2. To optimize the net updating parameters
3. To evaluate the net periodically
4. To snapshot the learned model and the solver state periodically

It is really important to understand how to use the (hyper) parameters defined on this file, in order to optimize the net, do it efficiently, test the net correctly and save the learned model in the correct time. Let us explain how to deal with it by an example of one of our solvers. The solver in particular is allocated in the path `models/googlenet_SR/solver.prototxt`.

4.2.3.1 Net definition, evaluation and snapshoting

Set the net definition indicating the path to the file "train_val.prototxt" to the parameter *net* in the solver. Set also the mode *solver_mode* to CPU or GPU, depending on the machine that we are using.

For that model our training dataset has $N = 116248$ images, and $M = 14540$ validation images. As we explained in section 4.2.2, the files containing the path and class for each image, are defined in this example in the file `models/googlenet_SR/train_val.prototxt` for both data layers, train and validation. Also in that file, we define the batch size for both layers as $bs_N = 32$ for the training dataset, and $bs_M = 32$ for the validation dataset.

With this information, we need to define:

- *test_iter*, as the number of forward steps that the net needs to test all validation images.
- *test_interval*, as the number of forward steps that the net needs to run over the training set, to start the testing on the validation images.
- *snapshot*, as the number of forward steps that the net needs to run over the training set, to save the the weights of the model in a .caffemodel file, and the state of the solver, in a .solverstate file.

i.e. After every *test_interval* training iterations, $test\_iter * bs_M$ validation images are fetched for testing.

The set up of *test_interval* really depends on how much tracking we want, in terms to see how the net works with data that is not being used for the training. We set this parameter in order to evaluate the net, every time that it has made the forward pass over the quarter of all training images. As the forward pass in the training it's done in a batch, the way to compute that is $test\_interval = N/4/bs_N$.

It is really important to save the model and the solver state, at the same time that performing a validation test. Otherwise we will not have the accuracy for the saved models in our log file, and later will be hard to choose the final model as we do in section 4.2.5. A way to do it, is setting up *snapshot* as a multiple of the value in *test_interval*. In our case, we set $snapshot = test\_interval * t$, for some $t \in \mathbb{N}$. We are aware that those files are heavy, so we set t not too small, on this case, $t = 6$. After that, we set also the path where we want to save this models, in the variable $snapshot\_prefix$.

To set *test_iter*, we just need to figure out how many forwards requires the validation test if it is done in batches of bs_M images out of M. i.e. $test\_iter = M/bs_M$.

So, we define this values as follows:

```
net: "models/googlenet_SR/train_val.prototxt"
solver_mode: GPU
test_iter: 454           # Number validation images 14540/32 = 454
test_interval: 908       # Number train images 116248/4/32 = 908
snapshot: 5448           # Save after a test interval. So this value has to be
                         # a multiple of the variable 'test_interval' (e.g.
    908*6=5448)
snapshot_prefix: "models/googlenet_SR/snapshots/ss_googlenet_SR"
```

Note that $32 * 454 = 14528 < 14540$, so the net will not test the entire validation set.

What can we do about that? Just set the bs_M in the file `models/googlenet_SR/train_val.prototxt`, to a value that has modulus zero with M, i.e. $M\%bs_M = 0$, as for example, set $bs_M = 4$. Then compute with the new value $test\_iter = M/bs_M$. The only issue with that, is that the testing step during the training will take longer, as usually the new value of bs_M will be lower. Otherwise we can just set $test\_iter = 455$ instead of 454, and the net will test couple of times $32 * 455 - 14540 = 20$ images.

4.2.3.2 Hyper-Parameters

We set the hyper-parameters described in section 2.2.3, following the strategy used by Krizhevsky et al. [16] in their famously winning CNN entry to the ILSVRC-2012 competition. This was exactly the parameters used to train the net googleNet, over the dataset imageNet in Caffe , i.e. the model that we are using for fine-tuning. The only differences in our definitions, is that we set η lower by a factor of 10, as we are not training from scratch. So the parameters are defined as follows:

```
base_lr: 0.001          # lr for fine-tuning lower than from scratch
lr_policy: "step"       # Learning rate policy: drop the learning rate in "steps"
                        # by a factor of gamma every stepsize iterations
gamma: 0.1              # drop the learning rate by a factor of 10
                        # (i.e., multiply it by a factor of gamma = 0.1)
stepsize: 20000         # drop the learning rate every 20K iterations
```

4.2.4 Train

After defining all files described above and setting the net parameters we perform the training phase of our project. We will use the default command of Caffe , but adding a pipeline to a file to save the output of the net, in order to plot the accuracy later. The required arguments for this command are, the solver, and the file .caffemodel containing the weights of the model that we are fine-tuning. The following command is used to execute the training phase and also to save the log, in the particular case of our model "googlenet_SR" from a snapshot of imageNet e.g. "bvlc_googlenet.caffemodel":

```
~/TFG$ caffe train -solver models/googlenet_SR/solver.prototxt -weights models/
    googlenet_SR/snapshots/bvlc_googlenet.caffemodel 2> models/googlenet_SR/LOGS/
    log.log
```

4.2.5 Choosing the best model

The training phase saves several .caffemodel files, which contains the learned weights. We use the best iteration, which is not necessarily the last one. Let us show how to find the best saved iteration automatically, and how to plot the net accuracy, using the default Caffe tools, gnuplot, and a script. We understand as best iteration or best model, the saved snapshot by Caffe with better accuracy on the validation set.

1. Parse the log created by the training to the file "log.log.test" into the current folder:

   ```
   ~/TFG$ python $CAFFE_ROOT/tools/extra/parse_log.py log.log $(pwd)
   ```

2. Plot the log test file generated by step 1:

   ```
   gnuplot
   reset
   set terminal png
   set output "googlenetACCURACY.png"
   set style data lines
   set key right
   set xlabel "iteratations"
   set ylabel "accuracy"
   set datafile separator ","
   plot "log.log.test" using 1:12 title "top-5", '' using 1:11 title "top-1"
   ```

A chart like the one above contains too much information to let us choose the best iteration. Also, as in the file `models/googlenet_SR/solver.prototxt` we set the parameter 'snapshot', just the iteration that are multiple to this variable are saved by Caffe . Thus, we choose the best .caffemodel based on the best iteration saved, following the script that we defined. Three parameters are needed:

- The file created by step 1: "log.log.test".

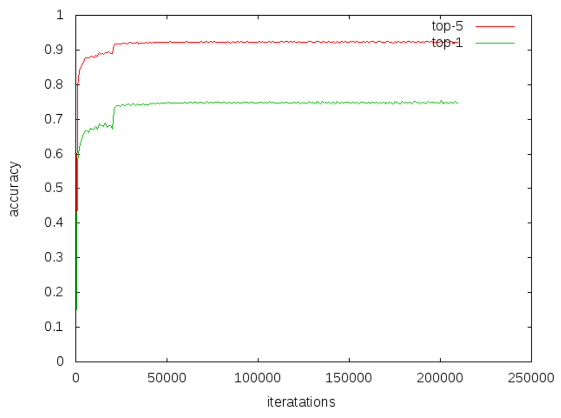

Figure 4.2: Example of how we show the accuracy vs the number of iterations performed.

- The columns we are interested to plot in that file above. First, we define the x-axis, that represents the number of iterations, and then a sorted y-axis, that defines the top-1 and top-5 accuracies.
- The parameter "snapshot", specified in the file `models/googlenet_SR/solver.prototxt`.

3. Get the best iteration. As an example we represent how we execute the script for googlenet model; we use column 1 for the x-axis, columns 11 and 12 as tops accuracies, and 2280 for the snapshot value.

```
~/TFG$ python tools/scripts/get_best_accuracy_iter.py 'log.log.test' 1
  11 12 2280
```

This will prompt the best iteration that corresponds to the one used as a final model. Also creates a .csv file that we can use to plot just the accuracies saved by the net. To do that, we just need to repeat step 2, this time with the new .csv file.

4.2.6 New Test Approach

For testing we use a python wrapper and a new approach that avoids preprocessing image manually and code duplication. The evaluation techniques are fully explained in section 5.2.

Once we have a trained model 'trained_model.caffemodel' for deploy, e.g. to classify an image or test the accuracy in our TEST dataset, so far, generally people create a 'deploy.prototxt' file as a copy of the file 'train_val.prototxt'. In that copy they switch the 'SoftmaxWithLoss' layer for a 'Softmax', in order to get the maximum argument in the probabilities vector, which gives us the predicted class, and replace the data layers for just a simple 'input' definition or 'input' layer[3].

Then a Matlab or Python wrapper is used to load an image or set of images and to run the net and classify. The obvious problem with it, is that we need to do the image preprocessing manually, so we need to be very accurate and use a procedure following exactly the parameters defined in the 'train_val.prototxt' file. Otherwise the results, for example, on the TRAIN or VAL dataset, would be different than in the training phase, and we do not want it to happen.

As we read so far, many people had many troubles with that. Also, we want to notice that deploying with the explained way, we are duplicating unnecessary code in our wrapper.

What we expose here is another variation to deploy the model that solves these problems: We just need to modify the data layer with phase 'TEST' in the 'train_val.prototxt' file, making the 'source' field pointing to our TEST dataset file 'test.txt' (see tree in 4.2.2), and rename this file to 'test.prototxt'. We can also remove the data layer with phase "TRAIN", but is not needed, because we are not using it.

Then we use a Python wrapper and we initialize the net with:

```
net = caffe.Net('trained\_model.caffemodel', 'test.prototxt', caffe.TEST)
```

After that, we just need to run the net to classify the first batch by:

```
net.forward()
```

And then we can get the probabilities vector for all the members of this batch using:

```
net.blobs['nameProbsLayer'].data
```

Where 'nameProbsLayer' is the name of the last inner product layer in our Net.

Finally, we want to let know to the reader, that the Caffe community is developing a way to train and deploy a model with a single .prototxt file definition, instead of the two described above. We can keep track of the issue in github[4].

4.2.7 Test

Once we have a trained model, we test the net using the metrics defined in section 5.2, with the new approach explained in section 4.2.6. To do so, we follow this instructions:

[3] https://github.com/BVLC/caffe/pull/3211
[4] https://github.com/BVLC/caffe/issues/3864

1. Add the new model information to the dict 'allModels' in the file `caffeWrapper.py`, filling the dict parameters as follows:

```
"new_model":
    {"caffemodel": 'models/new_model/snapshots/best_iter.caffemodel',
     "netDefinition":
         {"net_TEST": 'models/new_model/test.prototxt',
          "net_TEST_just_foodCAT": 'models/new_model/test_just_foodCAT.
    prototxt'},
     "nameLayer_AccuracyTop1": 'loss3/top-1',
     "nameLayer_AccuracyTop5": 'loss3/top-5',
     "nameLayer_innerProduct": 'loss3/classifier',
     "solver":                 'models/new_model/solver.prototxt'}
```

2. Run the test indicating the model and the dataset that we want to use. In this example we will use the model defined above and the balanced dataset (see section 2.1):

```
~/TFG$ python caffeWrapper.py -m 'new_model' -d 'net_TEST_balanced'
```

4.2.8 Portable Project

If we want a portable project, i.e. a CNN that can run easily on each computer with Caffe installed, we highly recommend using always relative paths in all files. If we do not, i.e. if we use absolute paths, then, we must change the paths in all files if we want to work with the net in another computer. Let us give a visual example about how to do it:

Go to the main folder:

```
~/TFG$ pwd
/home/pedro/TFG
```

Fine-tuning Imagenet for food recognition on *FoodCAT* dataset:

```
~/TFG$ caffe train -solver models/googlenet_SR/solver.prototxt -weights models/
  googlenet_SR/snapshots/bvlc_googlenet.caffemodel
```

This is how the parameters should look in "solver.prototxt":

```
~/TFG$ head -1 models/googlenet_SR/solver.prototxt
net: "models/googlenet_SR/train_val.prototxt"
```

This is how the parameters should look in "train_val.prototxt":

```
~/TFG$ grep source models/googlenet_SR/train_val.prototxt
source: "foodCAT_SR/train.txt"
source: "foodCAT_SR/val.txt"
```

This is how the file ”train.txt” should look:

```
~/TFG$ head -1 foodCAT_SR/train.txt
data/foodCAT_SR/panellets_de_codonyat/panellets_de_codonyat_354.jpg 0
```

As it is shown, everything is specified with relative paths from our main folder. It allow us to move our folder TFG to another computer and run it without any changes. It was spatially useful in this project, because we were working in local, but all computations run into the Peregrine HPC cluster explained in section 4.1.2.

4.3 Increasing the quality of the images by Super-resolution

In this section we show how we increase the resolution of the images over *FoodCAT* dataset using the method of SR [15]. To perform this task we use as a basis the python script that authors provide in their web site[5], and we change it to adapt it to our necessity.

When we fine-tune one of the models that we are using in Caffe, all images are resized to 256x256 as we explain in section 2.3. Therefore, we use SR to increase the size of each image in proportion to the times n, that the minimum dimension of the image (in width or height) needs to be increased to be bigger or equal to 256. n can be in the set $\{1, 2, 3, 4\}$, where $n = 1$, if the minimum dimension is already bigger or equal to 256, and $n = 4$, if the minimum dimension times 4 is equal or smaller than 256.

The script developed to do it is located in the path `TFG/tools/+tools/python_iccv/superresolution.py`, and it requires two parameters:

- Path ’p’ to the dataset.
- The minimum dimension ’R’ of the image.

The script works applying the SR to all the images for each of the folders in the dataset.

Below, we show the example of how we used:

```
~/TFG$ unbuffer python tools/+tools/python_iccv/superresolution.py -p 'data/
    images' -R 256 2>&1 | tee outfil
```

The commands ’unbuffer’ and ’tee’ are used to print the output of the script in the terminal and in the file ’outfil’ at the same time. It was done like this, because we can monitoring how the execution is going throw the terminal, and at the end, we can double check for errors in the file.

[5] http://www.ifp.illinois.edu/~dingliu2/iccv15/

Chapter 5

Results and Discussion

In this chapter we explain how we set the experiments, the obtained results, and which metrics are used for the evaluation, following the order of the objectives detailed in section 1.2. The objective to build a dataset is not considered as an experiment, but it is achieved and fully explained in section 3. In order to achieve the other objectives, we have developed several different networks, and we applied several different methods to the dataset, to improve the food recognizing task. The execution of each individual experiment and the evaluation of it, is just a particular case of the general method explained in section 4.2. Therefore, to recreate them, we just need to change the parameters that we provide with each experiment below, and repeat the steps described in section 4.2 with the new parameters.

5.1 Experiments Set Up

Each experiment explained below requires the specification of three sets of parameters:

1. The dataset, which we provide as folder containing the division in training, validation and test.
2. The model, which we provide as a folder containing the architecture of the net, the solver, and other parameters explained in section 4.2.2.
3. The weights of the model from where we are making the fine-tuning.

An example of these requirements with the dataset *FoodCAT*, and the model "googlenet_SR", is detailed visually in the tree of section 4.2.2. In practice, providing just the model and the weights, are enough to recreate the experiment, because the model itself is pointing to the dataset. In this section we provide the three set of parameters, in order to describe exactly how the experiment is built.

5.1.1 Food classifier

Let us first recall and describe each dataset we use, in order to reference them easily for each experiment. The given name, is also the folder name where we can find the dataset in the path `TFG`. After this, we expose the composition of each experiment, naming them as the real name we used, as a folder, inside the path `TFG/models`.

5.1.1.1 Datasets descriptions

All classes in the datasets below are extracted from the original datasets *Food-101* and *FoodCAT*, with the constraint that they must have at least 100 images. Otherwise, we do not include the class for the dataset:

- foodCAT_SR: consisting on the original dataset *Food-101*, and the dataset *FoodCAT* with the SR technique applied on it.
- foodCAT_SR_balanced: consisting on the dataset foodCAT_SR, but taking, at most, 500 images per class.
- foodCAT_resized: consisting on the dataset *Food-101* with all images resized to halved, and the original dataset FoodCAT
- foodCAT_resized_balanced: consisting on the dataset foodCAR_resized, but taking, at most, 500 images per class.
- foodCAT_OLD: consisting on the both original datasets *Food-101* and *FoodCAT*.
- foodCAT_500: consisting on the dataset foodCAT_OLD, but taking, at most, 500 images per class.

5.1.1.2 Experiments descriptions

The following experiments consist on fine-tuning the model specified below, with the dataset indicated, from a snapshot of imageNet. In training, we use the weights with path `TFG/models/model_name/snapshots/bvlc_googlenet.caffemodel`, where "model_name" is the name of the experiment that we are reproducing. Just in the case of VGG models, use the weights in path `TFG/models/model_name/snapshots/VGG_ILSVRC_19_layers.caffemodel`:

1. googlenet_SR: googleNet model with foodCAT_SR dataset.
2. googlenet_SR_balanced: googleNet model with foodCAT_SR_balanced dataset.
3. googlenet_resized: googleNet model with foodCAT_resized dataset.
4. googlenet_resized_balanced: googleNet model with foodCAT_resized_balanced dataset.
5. foodCAT_VGG_ILSVRC_19_layers: VGG model with foodCAT_OLD dataset.
6. foodCAT_VGG_ILSVRC_19_layers_500: VGG model with foodCAT_500 dataset.

The following fine-tuning are made from the snapshot of, fine-tuning GoogleNet model with *Food-101* dataset, from a snapshot of imageNet. In training, we use the weights with path `TFG/models/model_name/snapshots/foodRecognition_googlenet_finetunning_v2_1_iter_448000.caffemodel`, where "model_name" is the name of the experiment that we are reproducing:

7. googlenet_resized: googleNet model with foodCAT_resized dataset.
8. googlenet_resized_balanced: googleNet model with foodCAT_resized_balanced dataset.
9. foodCAT_googlenet_food101: googleNet model with foodCAT_OLD dataset.
10. foodCAT_googlenet_food101_500: googleNet model with foodCAT_500 dataset.

Above-mentioned, we have eight experiments using fine-tuning from the GoogleNet [16] model and just two from the VGG [17] model. Due to the lack of time, we decide to try out the experiments first with the GoogleNet model, and in future work, we will reproduce the experiments with the best performance, also with other models as VGG.

Note that experiment seven and eight are named equally as experiment three and four respectively. This is because the models three and seven are allocated in the same folder in the path `TFG/models/googlenet_resized`. The difference is that experiment three is launched from a snapshot of imageNet dataset, and experiment seven from a snapshot of *Food-101* dataset. Thus, the name of the file "solver.prototxt" is used for seven, and it is replicated and named with "solver_from_imagenet.prototxt" for experiment three, with the only difference in the parameter "snapshot", that in this case it points to the path `TFG/models/googlenet_resized/snapshots_from_imagenet` instead of `TFG/models/googlenet_resized/snapshots`, just to save the snapshots of both models in different folders. Same criterion is applied for experiment eight and four.

5.1.2 Categories classifier

This section shows how we set up two different experiments using the dataset *FoodCAT* in order to classify food dishes into one of the twelve categories defined in section 3.2.

The aim of these experiments is to build a model able to recognize food categories of different dishes with a high accuracy. To do it, we will fine-tune the GoogleNet CNN trained with the large dataset ImageNet. We will use this goal to study the network performance depending on if we train all layers or only the last, the fully-connected layer. The two experiments use:

- The dataset *FoodCAT* separated by categories located in the path `TFG/categories`.
- The GoogleNet model located in the path `TFG/models/googlenet_categories`.
- The weights from training GoogleNet model over the large dataset imageNet with path `TFG/models/googlenet_categories/snapshots/bvlc_googlenet.caffemodel`.

Once again, in practice, providing just the model and the weights, are enough to recreate the experiment, because the model itself is pointing to the dataset.

5.1.2.1 Fine-tuning just the fully-connected layer

The intuitive way to fine-tune a CNN only in the last layer (the fully-connected), is to set the learning rate η for all the other layers to zero. This can be done by Caffe, the framework that we are using, setting the parameter "lr_mult" to zero in all layers except the last inner product, which is the fully-connected layer that we want to train. The training model definition used for this experiment with these settings, can be found with the path `TFG/models/googlenet_categories/train_val_just_FC.prototxt`.

5.1.2.2 Fine-tuning all layers

This is the common experiment done on this project. Therefore, the file containing this training definition has the default name that we use for the training phase, "train.prototxt", located inside the model folder.

5.2 Evaluation Metrics

Many metrics can be considered to measure the performance of a classification task. In literature, mainly three methods are used: Accuracy Top-1, Accuracy Top-5, and the

Confusion Matrix. In real-world applications, usually the dataset contains unbalanced classes. The use of the above accuracies can hide the misclassification of the classes with fewer samples. e.g. consider a classifier with the predictions

$p = [\text{"}paella\text{"}, \text{"}peix\_al\_forn\text{"}, \text{"}paella\text{"}, \text{"}paella\text{"}, \text{"}peix\_al\_forn\text{"}, \text{"}peix\_al\_forn\text{"}]$,

and the respectively true labels

$t = [\text{"}paella\text{"}, \text{"}peix\_al\_forn\text{"}, \text{"}paella\text{"}, \text{"}paella\text{"}, \text{"}peix\_al\_forn\text{"}, \text{"}sopa\_de\_rap\text{"}]$.

As we can see, the class "sopa_de_rap" has just a single sample, and the classifier can not predict it at all, while the accuracy Top-1 gives us $\frac{5}{6} * 100 = 83.33\%$. Thus, we consider a Normalized Accuracy Top-1, explained below, that give us the information on how good the classifier is no matter how many samples each class has. In this example, two of the classes are perfectly predicted and one not at all, i.e. $NAT1 = \frac{2}{3} * 100 = 66.66\%$.

5.2.1 Formal definitions

Let us define formally each metric with the next notations:

Let N be the total number of classes with images to test, let N_i be the number of images of the i-th class, and set $n = \sum_{i=0}^{N-1} N_i$, as the total number of images to test.

Let $\hat{y}_{i,j}^{k}$ be the `top-k` predicted classes of the j-th image of the i-th class, and $y_{i,j}$ the corresponding true class.

Let us also define $\mathbf{1}_A \colon X \to \{0,1\}$ as the indicator function by

$$\mathbf{1}_A(x) := \begin{cases} 1 & \text{if } x_i \in A \text{, for some } i, \\ 0 & \text{if } x_i \notin A \text{, for all } i. \end{cases}$$

Then, the definitions of the metrics are as follows:

5.2.1.1 Accuracy Top-1

$$\texttt{AT1} = \frac{1}{n} \sum_{i,j} 1_{y_{i,j}}(\hat{y}_{i,j}^{1}) \tag{5.1}$$

5.2.1.2 Accuracy Top-5

$$\texttt{AT5} = \frac{1}{n} \sum_{i,j} 1_{y_{i,j}}(\hat{y}_{i,j}^{5}) \tag{5.2}$$

5.2.1.3 Normalized Accuracy Top-1

$$\texttt{NAT1} = \frac{1}{N}\sum_{i=0}^{N-1}\frac{1}{N_i}\sum_{j=0}^{N_i-1} 1_{y_{i,j}}(\hat{y}^1_{i,j}) \tag{5.3}$$

5.2.1.4 Confusion Matrix

By definition a confusion matrix CM is such that $CM_{i,j}$ is equal to the number of observations known to be in class i but predicted to be in group j, where i are rows and j columns.

Using the example of predictions p and true labels t introduced at the beginning of this section, the associated CM looks as follow:

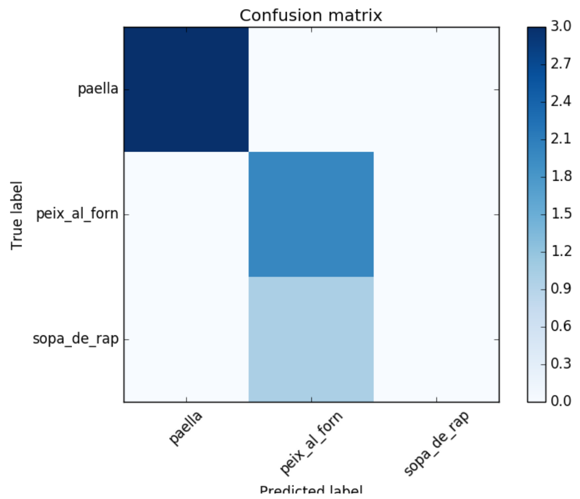

FIGURE 5.1: Visualizing a CM for a three classes classification

5.2.2 Metric relative to ***FoodCAT*** dataset

We are classifying food images from two datasets, *Food-101* and *FoodCAT*. *FoodCAT* dataset is not balanced and most of the classes have the half of images that *Food-101* have. Thus, it is also interesting to measure how the CNN models works with just the *FoodCAT* dataset. Therefore, we are using the measurements explained above with the entire dataset, but also with just one of them; the *FoodCAT* dataset. Furthermore, this is the reason why all datasets described in section 5.1.1.1 have a file named "test_just_foodCAT.txt", and why all models described in section 5.1.1.2 have a file named "test_just_foodCAT.prototxt". These files make possible this test.

We keep in mind that our balanced datasets have classes with 500 images at most, and *FoodCAT* has in average 400. Then, when we balance *FoodCAT*, the resulting dataset is almost equal to the original. In fact, original *FoodCAT* has 44304 images and after the balance 40852. Which makes it possible to compare the accuracy over the *FoodCAT* dataset between models using balanced or not balanced datasets. On the other hand, we can not compare the global accuracy, because in balanced datasets the testing is made with half of the images than in a not balanced dataset, so it wont be a good measure.

5.3 Results

In this section we present the obtained results following the evaluation metrics described in section 5.2 for each of the experiments developed and described in section 5.1.1.2. It is divided in two sections: Food recognizer, and categories recognizer.

5.3.1 Food recognizer

In this section we present the arguments for the next two conclusions:

1. Balanced classes during the training phase are really important to recognize, with similar accuracies, different datasets with a single CNN. Even if using balanced classes means decrease the number of samples in each class in one of the datasets, in order to balance with the one who has fewer images per class.
2. The Super-resolution method over the *FoodCAT* dataset improves the accuracy of recognizing food dishes.

5.3.1.1 Effects of reducing the number of classes on the larger dataset *Food-101*

Let us first present the table 5.1 summarizing all target results for each experiment from seven until ten. We want to recall that each of these experiments are a CNN fine-tuning for the datasets *Food-101* and *FoodCAT* together, using the weights from a previous fine-tuning for the dataset *Food-101*, which was self developed using the weights from a CNN trained for the large imageNet dataset.

The performance of these tests is not used to measure how good our system is recognizing food dishes. This is because the test dataset used for these experiments can contain images used for the training of the CNN weights, which we are using for fine-tuning our models. Nevertheless, these experiments were driven to test how the accuracy changes, when we apply a balance for all classes.

Experiment	7		8		9		10	
Datasets	A, B	B	A, B	B	A, B	B	A, B	B
$AT1$	75.37	48.64	68.29	48.33	**73.55**	45.85	69.67	**51.60**
$AT5$	**92.42**	81.93	89.52	80.94	91.59	79.46	90.42	**83.06**
$NAT1$	63.44	42.70	62.91	44.06	61.06	39.19	**64.33**	**46.84**

TABLE 5.1: The results of the experiments from 7 to 10, described in section 5.1.1.2. A=*Food-101*, B=*FoodCAT*. Best results are shown in boldface.

For experiments seven and eight in table 5.1, we use the original *FoodCAT* dataset, and the dataset *Food-101* with the resolution of all images halved. It is used like this, because the average image resolution of *Food-101* doubles the average in *FoodCAT*, and after to have applied the resizing, the average gets closer (see section 3.1.4.2 for more details). Experiment nine and ten are performed for the original datasets.

Table 5.1 is organized by the experiment combined columns and the dataset columns, describing the used dataset ('A, B' when it is over both datasets, and 'B' it when is just for *FoodCAT*). We set the best $AT1$, $AT5$, and $NAT1$ in bold, for each of the tested datasets (*Food-101*+*FoodCAT* or *FoodCAT*). We can see that the best results for the dataset *FoodCAT* (columns 'B'), are better achieved by experiment ten, which is 'foodCAT_googlenet_food101_500', the CNN trained from the original datasets with balanced classes. Thus, we can say that for recognize just the healthy dataset *FoodCAT*, with less images than *Food-101*, it is better to reduce the *Food-101* dataset to have a similar number of classes than *FoodCAT*. On the other hand, the results of the test in both datasets together (columns 'A, B'), are better when we use all samples in both datasets (columns 7 and 9), because those compared with the balanced datasets, we are getting twice as many samples in Food-101 during the learning epoch, which is what we expected.

5.3.1.2 Effects of applying the super-resolution and other resizing methods on the dataset *FoodCAT*

Motivated by the results of the previous section, where the best performance to recognize the *FoodCAT* dataset is achieved, by the CNN trained reducing the number of samples in the larger dataset *Food-101*. We compare how resizing methods change the capability of the CNN to recognize all food dishes. Unlike in the previous section, all models here are fine-tuning using the weights of the larger dataset imageNet. Therefore, all images used for testing are the first time that the CNN classifies them. Hence, the results of this section are finally used to measure the final accuracy of our system to recognize food classes.

Experiment	1		2		3		4		5		6	
Datasets	A, B	B	A, B	B	A, B	B	A, B	B	A, B	B	A, B	B
$AT1$	**68.07**	50.02	62.41	48.94	67.16	49.66	61.28	48.85	67.74	48.12	65.16	**50.59**
$AT5$	**89.53**	81.82	86.81	81.63	89.27	82.07	86.52	80.92	89.28	81.03	88.94	**83.40**
$NAT1$	59.08	44.25	57.91	44.44	58.57	44.31	56.99	44.44	58.18	42.34	**60.74**	**46.53**

Table 5.2: The results of the experiments from 1 to 6, described in section 5.1.1.2. A=*Food-101*, B=*FoodCAT*. Best results are shown in boldface.

Table 5.2 is organized by the experiment combined columns and the dataset columns, describing the used dataset ('A, B' when it is over both datasets, and 'B' it when is just for *FoodCAT*). We set the best $AT1$, $AT5$, and $NAT1$ in bold, for each of the tested datasets (*Food-101*+*FoodCAT* or *FoodCAT*). Again, we can see that the best results for the dataset *FoodCAT* (columns 'B'), are better achieved by a CNN trained from the original datasets with balanced classes (experiment 6: 'foodCAT_VGG_ILSVRC_19_layers_500'). Once again, it shows the importance of the balanced classes to recognize, with similar accuracies, different datasets with a single CNN. Furthermore, the results of the test in both datasets together (columns 'A, B'), are better when we use all samples in both datasets during the training phase, with the method SR applied for *FoodCAT*. This CNN is the one used in experiment 1, and it also achieves the second best result for the $AT1$ over the *FoodCAT* dataset, with a score of 50.02, just 0.57 less than the balanced datasets with VGG (experiment 6). Moreover, adding all scores for the accuracies $AT1$ and $AT5$, over the two tests 'A, B' and 'B', experiment 1 has the highest value with 289.44 followed by experiment 6 with 288.09. Experiment 1 has also the second best $AT1$ over the *FoodCAT* dataset.

With all this data, we choose GoogleNet as the best model, trained from all samples of both datasets, with the SR method applied for *FoodCAT*, corresponding to experiment 1. In our future work, we will train a mix over the two winners for these tests: the VGG model with the SR method applied for *FoodCAT* and both balanced datasets.

We want to note these as the best models, the foodCAT_VGG_ILSVRC_19_layers_500 and googlenet_SR, corresponding to experiments 1 and 6 respectively, during the training phase, the models continued to learn the last iteration, i.e. the last iteration has the best performance. Figure 5.4 shows the accuracies over the validation set during the training phase. Therefore, if we let the models learn longer, the nets should get a better performance.

Figure 5.5 shows the CM for googlenet_SR model. Taking a closer look of the upper left quarter (*FoodCAT* classes), we see that it is more diffuse than the bottom right (*Food-101* classes). That is because performance over *Food-101* is better than *FoodCAT*. Moreover, we observe that the net separates very well the two datasets, as there are not many predictions from one dataset to the other. In the future work, we will deeply study

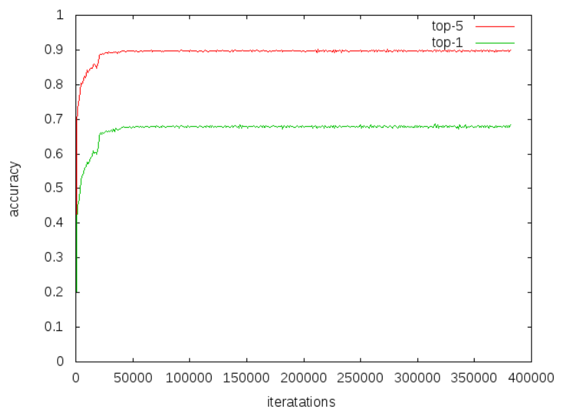

FIGURE 5.2: Accuracy for googlenet_SR model

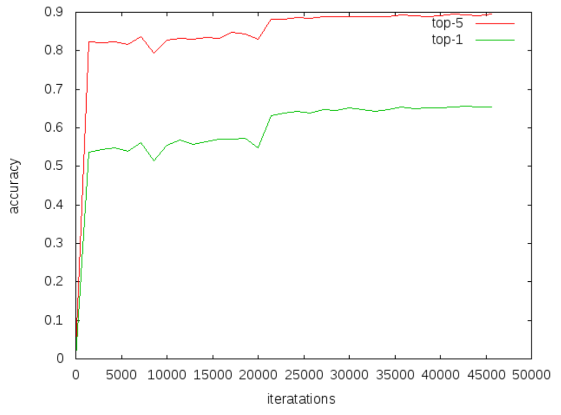

FIGURE 5.3: Accuracy for VGG balanced model

FIGURE 5.4: The accuracy over the validation set during the training phase for models googlenet_SR and foodCAT_VGG_ILSVRC_19_layers_500, respectively. We can see that the net was learning until the last iteration for both models.

this phenomena in order to know if the CNN is really understanding both datasets as different food clusters, or if it is more related to the resolution of the images. Due to that, *Food-101* images always have weight or height equalling to 512.

5.3.2 Categories recognizer

In this section, we expose the best result obtained in this project. It is a novel way to classify food dishes automatically into one of the twelve categories defined in section 3.2. A complete net fine-tuning and a fine-tuning for only the last fully-connected layer are compared.

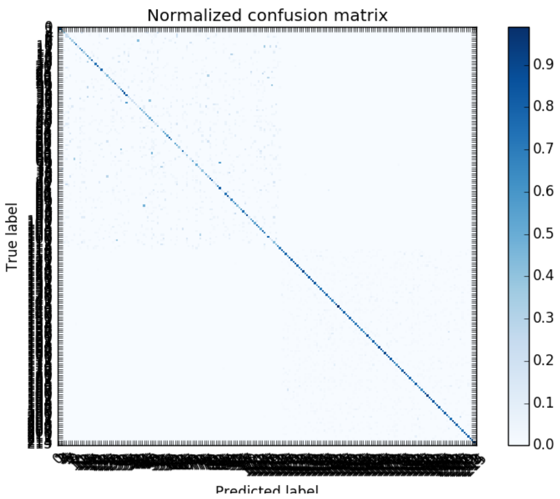

Figure 5.5: *CM* of googlenet_SR model. We remark that the upper left quarter (*FoodCAT* classes) is more diffuse than the bottom right (*Food-101* classes). It is because performance over the second is better than for the first.

5.3.2.1 Learning time comparative among training all layers or only the fully-connected

As expected, training only the FC layer is faster and less effective than training all layers of the CNN. theoretically, it is explained because in the case of all layers there are many more parameters to learn. Figure 5.8 shows in practice how the AT1 evolves for each iteration over the models trained for all layers (figure 5.6), and only on the FC (figure 5.7).

The training phase is executed on a cluster during a limited time, that for this case was set to 24 hours (see section 4.1.2). Also, both experiments were configured to train at most during 1.000.000 iterations. In Table 5.3, the last three columns show the number of iterations that the models were training during a certain period of time, and the best iteration, in terms of AT1 over the validation set during the training. The FC model completed all training twice as fast than the "All layers" model, and both models show the best iteration twenty times faster than the maximum iterations, which gives us the tip that not too many iterations were needed in order to learn this model.

Obviously, a result like this would not be possible without the fine-tuning. The time required to train a model from scratch always depends on the used dataset, but is not shorter than three days in a good GPU as we are using.

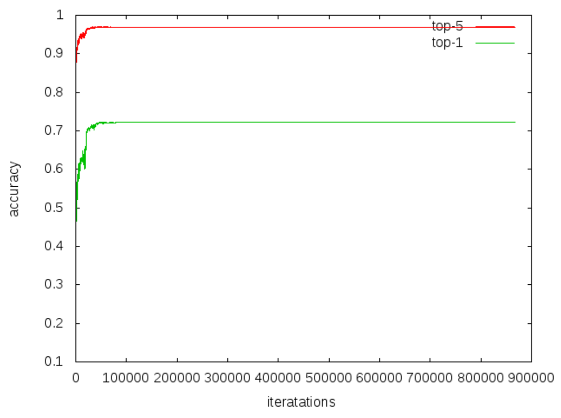

FIGURE 5.6: GoogleNet training all layers

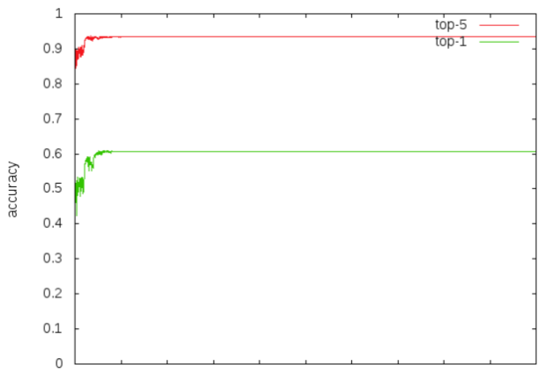

FIGURE 5.7: GoogleNet training only the fully-connected layer

FIGURE 5.8: Figures 5.6 and 5.7 show the accuracy top-1 over the validation set during the training phase.

5.3.2.2 Performance classifying a food dish into a food category

In table 5.3 we provide the results obtained for this task. First, if we have a limited machine or limited time, we show that fine-tuning just the fully-connected layer over a model previously trained on a large dataset as imageNet [18], can give a good performance. This is a good and interesting point because GoogleNet model trained on ImageNet is able to classify 1.000 classes, where just two, 'snail' and 'mushroom', belong to our categories. As our propose is to use this work on a medical system with real patients, we require better performance. Training all layers, we set the novel state of the art recognizing food categories over Mediterranean food as it is shown in the table, with $AT1 = 72.29$, and $AT5 = 97.07$. Taking care of the difference of samples on each class, the normalized measure also gives a high performance, with $NAT1 = 65.06$.

Figure 5.9 shows the normalized confusion matrix for GoogleNet model trained over all layers. It is not surprising that 'postres y dulces' is the category that the net can recognize better, as it is also the class with more samples in the dataset with 11933,

	AT1	AT5	NAT1	# Iterations	Best iteration	Time executing
FC	61.36	93.39	50.78	1.000.000	64.728	12h
All layers	**72.29**	**97.07**	**65.06**	900.000	49.104	24h

Table 5.3: Performance and learning time, fine-tuning the GoogleNet model over the categories dataset. We show the results for two experiments done; training all layers, and only training the last fully-connected. Best results are shown in boldface.

followed by 'carnes' with 7373 (see section 3.2 for more information about this dataset). The classes with less samples in our dataset are 'setas' and 'caracoles', but those specific classes are respectively 'mushroom' and 'snail', the ones that also imageNet contains (the dataset used for the pre-trained model that we are using). We believe that this is the reason that makes this classifier still good for these classes.

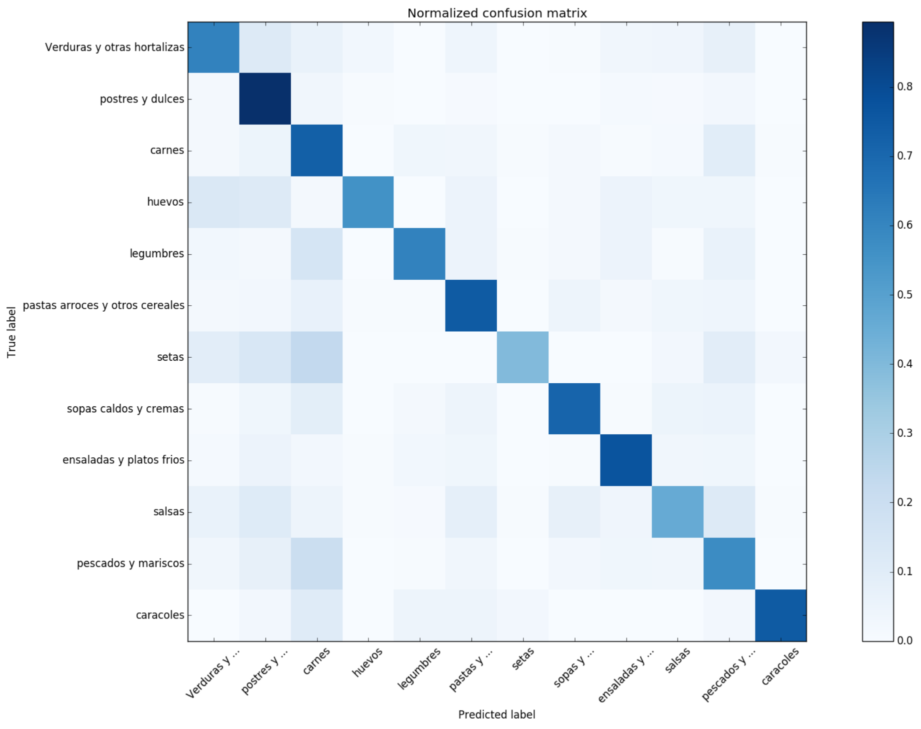

Figure 5.9: Normalized CM of GoogleNet model trained over the all layers to recognize food categories.

5.4 Discussion

In this section we explain the advantages and disadvantages that we found during the implementation of the project. Mainly, we expose the hard work that to build a new dataset represent, the benefits of fine-tuning pre-trained models, and the expensive cost to use a CNN.

5.4.1 Advantages and Disadvantages

When we want to face an image recognition problem with a specific topic, in our case, food dishes, and we want to perform it using a supervised CNN, we need a clean and large dataset. In our proposed, the target dataset is a food dataset related to Mediterranean diet, and it was not built before, so we did. The first part, scrap websites and download the images is funny and motivating. Later, as we did not have a system to know if the downloaded images were truly related to the class that they were representing, we checked all images for each class one by one manually, and this is really a boring and exhausted step. The good news is that with this project we built a system that can do this task automatically with a very high precision. In future work, to extend the dataset we will use our CNN classifier to perform this task lighter.

Other disadvantages building the dataset is that the classes were not balanced and the images present very different resolutions. This features shown to be relevant when we are training a classifier as CNN, and our results shown that methods as the SR to increase the resolution of the pictures, helps to get a better accuracy.

Once we have a dataset, probably it has many pictures per class, but not as many as others datasets already public. When we are using a CNN to recognize classes, as many pictures we have, better the accuracy will be. Our work shows, that using the weights of a pre-trained model over a larger dataset, even if the datasets are different, we can obtain very good accuracies recognizing our particular dataset. The disadvantage about it, is that training a CNN requires a good GPU, and it is a expensive PC component.

5.4.2 Advices

We found that the way to enjoy more facing a image recognition problem, is to drive it to a subject that has already available datasets to use. Therefore, we can put all our effort just in the preprocessing and the recognition task.

Before start a project that involves a CNN, we found the constraint to have a good computer with a good GPU, at least as described in section . At the beginning of this project we just had a conventional laptob with a CPU *Intel Pentium(R) Dual CPU T3200@2.00GHzX2* , and a graphic card *Mobile Intel GM45 Express Chipset.* The GPU did not support well Caffe, so we used on the CPU, and training only 3.000 iterations took two days.

Chapter 6

Conclusion and Future Work

In this chapter we conclude the thesis. First, we summarize the project contributions and later we expose the future roadmap.

6.1 Conclusions

Below, we list the contributions of the project to the field of food image classification:

- We present the **novel and challenging multi-labeled dataset related to Mediterranean diet called *FoodCAT***:
  - For the first kind of labels, the dataset is divided into **115 food classes** with an average of 400 images per dish.
  - For the second kind of labels, the dataset is divided into **12 food categories** with an average of 3800 images per dish.
- **Recognizing food classes**, the best model has been obtained training from the datasets ***FoodCAT***, after **increase the resolution** with the **novel method Super-resolution**, and ***Food101***. This model achieves the highest accuracies:
  - **top-1 with 68.07%** and **top-5 with 89.53%**, testing both datasets together.
  - **top-1 with 50.02%** and **top-5 with 81.82%**, testing only ***FoodCAT***.
- **Recognizing food categories**, we achieve the highest accuracies **top-1 with 72.29%** and **top-5 with 97.07%**.
- We had a relevant involvement with the Caffe community (see appendix A):

- We introduced a new method to test the CNN avoiding code duplication and manual image preprocessing.
- We uploaded the model definition for the Net VGG.
- We presented a friendly step-by-step guide to use the framework with developed scripts automating tasks.

6.2 Future work

Through all the research, we would like to emphasize the fact that the healthy dataset *FoodCAT* could be enlarged, in order to publish it and let the people work with it, with the common objective to improve people's life regarding food problems. This work is already started and we currently are using social networks as Instagram[1] or foodspotting[2].

Secondly, we will integrate our food recognizer system into the start up Onfan[3], who requires it, from the beginning of the project. Onfan is a new gastronomic guide that works as a social network, feeding on users contributions. It works like a mixture between tripadvisor and Instagram, because it is a guide based on pictures, very visual and intuitive. The food images are uploaded and tagged by the users with a giving name. Now, with our new development, the users will enjoy with an automatic tagging for more than 200 food dishes.

Once we enlarged the dataset, we will create a new model able to recognize a healthy diet based on the Mediterranean food. For that task, as we had mentioned in section 5.1.1.2, we base our work on a set of models, such as googleNet, VGG, resNet [17], and others.

Finally, we will integrate our system with the new accepted project "Life-logging based environment for holistic dietary pattern and lifestyle assessment for health status biomarkers identification and validation" of the CVUB research group, that belongs to the Department of Mathematics and Computer Science of the University of Barcelona.

[1] `https://www.instagram.com/`
[2] `http://www.foodspotting.com/find/in/The-World`
[3] `https://www.youtube.com/watch?v=jvb_DW2kL-w`

Appendix A

Caffe contributions

On this appendix, we summarize our contributions to the Caffe community.

- New Test approach (see section 4.2.6). Link to caffe users. One month after the publication, this is the traffic of the post:

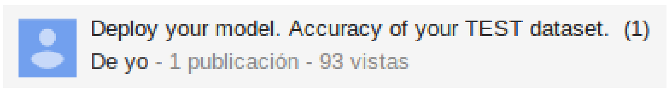
Deploy your model. Accuracy of your TEST dataset. (1)
De yo - 1 publicación - 93 vistas

FIGURE A.1: Traffic in Caffe Users of our new Test Approach

- VGG net definition (see section 2.3.2). Link to github. We can see our reference at the end of the page. One month after the publication, this is the traffic of the files:

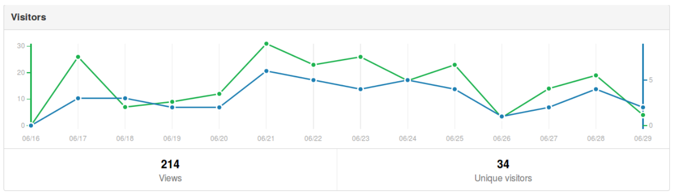

FIGURE A.2: Traffic in GitHub of our VGG definition

- Question in Caffe users answered by us.
- Question in Caffe users answered by us.
- Question in Caffe users answered by us.

Bibliography

[1] F. Monteiro-Silva. Olive oil's polyphenolic metabolites - from their influence on human health to their chemical synthesis. *ArXiv e-prints*, January 2014.

[2] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014. URL http://arxiv.org/abs/1409.4842.

[3] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

[4] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[5] Lukas Bossard, Matthieu Guillaumin, and Luc Van Gool. Food-101 – mining discriminative components with random forests. In *European Conference on Computer Vision*, 2014.

[6] Atsushi TATSUMA and AONO Masaki. Food image recognition using covariance of convolutional layer feature maps. *IEICE TRANSACTIONS on Information and Systems*, 99(6):1711–1715, 2016.

[7] Marc Bolaños and Petia Radeva. Simultaneous food localization and recognition. *CoRR*, abs/1604.07953, 2016. URL http://arxiv.org/abs/1604.07953.

[8] Institut Catalá de la Cuina. *Corpus del patrimoni culinari catalá*. Edicions de la Magrana, 2011. ISBN 9788482649498.

[9] Hajime Hoashi, Taichi Joutou, and Keiji Yanai. Image recognition of 85 food categories by feature fusion. In *ISM*, pages 296–301. IEEE Computer Society, 2010. ISBN 978-1-4244-8672-4. URL http://dblp.uni-trier.de/db/conf/ism/ism2010.html#HoashiJY10.

[10] Taichi Joutou and Keiji Yanai. A food image recognition system with multiple kernel learning. In *Proceedings of the 16th IEEE International Conference on Image Processing*, ICIP'09, pages 285–288, Piscataway, NJ, USA, 2009. IEEE Press. ISBN 978-1-4244-5653-6. URL http://dl.acm.org/citation.cfm?id=1818719.1818816.

[11] Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. Deep image: Scaling up image recognition. *CoRR*, abs/1501.02876, 2015. URL http://arxiv.org/abs/1501.02876.

[12] Kevin Gurney. *An Introduction to Neural Networks*. Taylor & Francis, Inc., Bristol, PA, USA, 1997. ISBN 1857286731.

[13] Yann LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Effiicient backprop. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, pages 9–50, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-65311-2. URL http://dl.acm.org/citation.cfm?id=645754.668382.

[14] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman. Return of the Devil in the Details: Delving Deep into Convolutional Nets. *ArXiv e-prints*, May 2014.

[15] Zhaowen Wang, Ding Liu, Jianchao Yang, Wei Han, and Thomas Huang. Deep networks for image super-resolution with sparse prior. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 370–378, 2015.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E.Hinton. Imagenet classification with deep convolutional neural networks. In F.Pereira, C.J.C.Burges, L.Bottou, and K.Q.Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

[18] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.