University of St Andrews

MSc Applied Stats & Data Mining

Machine Vision: theory and application on biological data

Alberto Ferrando

supervised by
Dr. Carl Donovan

August 29, 2016

Declarations:

I, Alberto Ferrando, hereby certify that this thesis, which is approximately 14,000. words in length, has been written by me, and that it is the record of work carried out by me, or principally by myself in collaboration with others as acknowledged, and that it has not been submitted in any previous application for a higher degree.

I was admitted as a post-graduate student in September 2015 and as a candidate for the degree of MSc in Applied Statistics & Data Mining in August 2016.

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and the abstract may be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that my thesis will be electronically accessible for personal or research use and that the library has the right to migrate my thesis into new electronic forms as required to ensure continued access to the thesis.

Date 29 August 2016 signature of candidate

Contents

1 Introduction

Pattern recognition of images refers to the process of building an algorithm capable of automatically classify images based on their content. Also called *Machine Vision*, it has important industrial applications the most notable (and recent) of which are driver-less cars. Such domain is intensely interdisciplinary, taking advantage of computer science and statistical methodologies. Indeed, to be able to correctly classify images, it is not only important to construct a well performing statistical model, but also understand how the images were collected, how they are stored on a memory and how they can be represented in mathematical forms. All this knowledge is fundamental given that images cannot be passed as they are (*i.e.* in raw form) but they must be processed to eliminate noise, extract key regions or objects and also to extract *variables* or *features*: numerical summaries of the objects that are used to describe them while minimizing the amount of data passed to the statistical model.

In this dissertation, this whole process will be described by addressing the most important techniques & methodologies. First, a discussion on image acquisition and mathematical representation followed by the presentation of processing algorithms such as filters and point operations. The subsequent section will describe features and their extraction. The dissertation will then venture into the statistical procedures used for classification, in particular neural networks and their architecture. Finally, an application of image pattern recognition will follow in which all the theory explained in previous sections will be applied.

Image formation and feature extraction

The first steps towards automatic classification of images are to correctly process the images in a machine-readable form and to extract important *features* from them. The latter is of paramount importance: it entails the selection and extraction of a distinguishing primitive characteristic or attribute of an image [1], *i.e.* a measured variable intended to be informative and non redundant. Such features could be subsequently used as independent variables in a classifier. In this section, I will present the basic ideas behind image formation and transformation. In particular, the theory behind the specific features used in this application will also be presented.

2 Image acquisition

The process that transform a visual scene into a digital representation is quite complex and its detailed description lies outside the scope of this document (for a more detailed description see [2] and [3]). Nevertheless, all image acquisition methods basically create a two-dimensional, time-dependent and continuous distribution of light radiation. To achieve this, three steps are required:

1. The light must be *spatially sampled*
2. The resulting discrete distribution must then be *temporally sampled*
3. The values obtained must then be translated in machine-readable form, a process called *quantization*

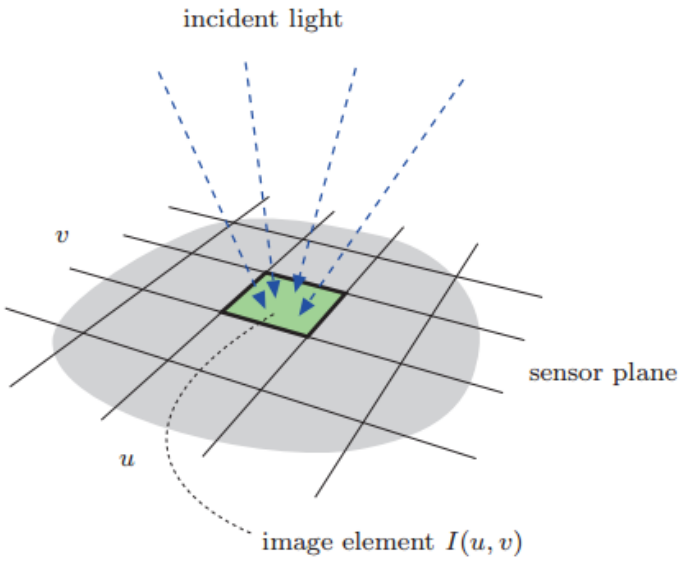


Figure 1: Schematic representation of the sensor/image plane of the acquisition tool. By discretizing the signal, its geometry defines the spatial sampling of the continuous image. u and v represent the columns and rows respectively. Credits to [2]

.

The first step transforms the continuous signal into a discrete one and is carried out by the acquisition device (*e.g* the digital camera). How is done depends on the geometry

148	123	52	107	123	162	172	123	64	89
147	130	92	95	98	130	171	155	169	163
141	118	121	148	117	107	144	137	136	134
82	106	93	172	149	131	138	114	113	129
57	101	72	54	109	111	104	135	106	125
138	135	114	82	121	110	34	76	101	111
138	102	128	159	168	147	116	129	124	117
113	89	89	109	106	126	114	150	164	145
120	121	123	87	85	70	119	64	79	127
145	141	143	134	111	124	117	113	64	112

Figure 2: Matrix representation of a 10x10 image.

of sensors of such acquisition device: usually they are arranged on a rectangular plane as depicted in Fig.1, where each square sensor register the signal of the incident light. Subsequently, the amount of light reaching each of those sensors is *temporally sampled* by measuring it at regular intervals of time: the process is essentially carried out through electric and physical processes. Finally, the values of this light sample are *quantized*, that is transformed into a matrix of values, more commonly integers on a given scale (*e.g.* from 0 to 255, white to black). Floats can also be used. This conversion is carried out by specialized hardware called *analog-to-digital converter*.

The result of this process is a two-dimensional matrix of values (pixels) as showed in Fig.2. Formally, a digital image is a function of coordinates $\mathbb{N}$x$\mathbb{N}$ that maps to a set of possible pixel values $\mathbb{P}$:

$$I(u, v) \in \mathbb{P} \qquad \text{and} \qquad u, n \in \mathbb{N} \tag{1}$$

The *size* of an image is therefore defined by u and v, that are the number of columns and rows of the image I. The concept of image *resolution* is consequently linked to its size: in fact, it refers to the number of Picture Elements (pixels) per measurement, for example PPI (pixels per inch). Therefore, the higher the resolution, the clearer and more detailed the image will be. In particular, it is worth noting that the term resolution is sometimes used wrongly[1] to define the pixels count: this can create confusion, given that the resolution is not only dependent on the number of pixels, but also on their physical

[1] Guideline for Noting Digital Camera Specifications in Catalogs

size. Nevertheless, for image classification, the image resolution is not important: what really matters is the size of the image matrix.

2.1 Pixel values

The information contained in a pixel always depend on the data type that contains. Pixel values are binary words of length k so that 2^k values can be represented: thanks to this, multiple types of images can be stored. Here are presented the most common one: *grey-scale*, *binary* and RGB.

Binary Images
Binary images are a special case of grey-scale. In them, the pixels can only take two values [0;1], black or white (or any other colour couple). In the context of image classification, they arise as products of procedures such as segmentation and thresholding.

Grey-scale Images
A grey-scale image is represented by a single channel (*i.e.* a single matrix) that encodes the brightness or intensity of the image. Typically, only positive integers are used, hence pixel can take values $[0 \ldots 2^k - 1]$. Usually, this type of images use 8 bit per values *i.e.* $k = 8$ even though for many specialized and professional applications 12 to 16 bits per pixels are required.

Colour Images
Colour images are encoded through multiple different channels, depending on the colour scheme used. Most commonly, the RGB (Red-Green-Blue) colour model is used: this entails 3 channels for each pixel, that is one channel for each colour component. Therefore, any given pixel will have three integer values between [0, ..., 255] for a total of $8x3 = 24$ bits. Other colour models exists, which obliviously entail a different number of colour channels: *e.g.* the CMYK (Cyan-Magenta-Yellow-Black) carries 4 different values for every single pixel in the image.

Special Images
Sometimes, specific applications require a particular digital translation of the image and none of the above cases would suffice. The two most common special cases are images with negative values and images with floating-point values. The former arise in pre-processing steps, for example after filtering. On the contrary, the latter can be found in cases when

great precision and detail are required, for example in medical imaging or astronomical research.

3 Image Pre-processing

Before extracting the features, an image must be pre-processed in order to decrease noise and separate the relevant objects from the irrelevant ones. Several different techniques of image processing exist, each one with its own purpose and application domain. Most of the techniques in image processing are in fact independent from pattern recognition, and have been devised to deal with a plethora of different tasks. In this paper, only the methodologies used for the application will be presented. In particular *point operations*, *filters* and *fore/background separation*.

3.1 Point Operations

Point operations refer to all those modifications applied to single pixel values without affecting neither the geometry/size nor the image local structure. This is possible because each new pixel value $\alpha' = I'(u, v)$ is only dependent on the original value of the pixel at the same coordinates $\alpha = I(u, v)$ and independent from the neighbours[2]. Therefore, a point operation can be formally described as:

$$\alpha' = f(\alpha) \qquad \text{or} \qquad I'(u, v) = f(I(u, v)), \qquad \forall\ (u, v) \tag{2}$$

If the function $f(\cdot)$ does not depend on the coordinates (u, v) (*i.e.* it's the same for the whole image matrix), we call it an *homogeneous* point operation. Typical examples:

- Intensity transformation
- Brightness transformation
- Thresholding
- Gamma correction

For our purpose, the thresholding will be explored further. For all the others point operations, the interested reader is referred to [2].

[2] if it's not independent, we call the operation a *filter*: see next section

For completeness, it is also interesting to note the existence of *non-homogeneous* point operations, in which the function $g(\cdot)$ would also depend on the pixels coordinates:

$$\alpha' = g(\alpha, u, v) \qquad \text{or} \qquad I'(u,v) = g(I(u,v), u, v), \qquad \forall\ (u,v) \tag{3}$$

A common example of non-homogeneous point operation is local light/contrast adjustment to compensate for uneven lightning during image acquisition.

3.1.1 Thresholding

Thresholding an image means separating the pixel values in to classes depending on a given constant value a^*: hence, the thresholding function $f_{threshold}(\alpha)$ reduces the pixels to two fixed intensity values a_0 and a_1:

$$f_{threshold}(\alpha) = \begin{cases} \alpha_0, & \text{if } \alpha < a^* \\ \alpha_1, & \text{if } a \geq a^* \end{cases} \tag{4}$$

with $0 < a^* \leq a_{max}$. The most common thresholding operation is *binarization* which happens when $a_0 = 0$ and $a_1 = 1$, producing a binary image. Binarization is a simple yet effective procedure to reduce noise in the image and separate interesting regions/objects from the background.

3.2 Filters

Filters, differently than point operations, uses multiple pixels sources to compute the new pixel values without affecting the size of the image. The best example is *smoothing* an image. Such filter substitutes each pixel value by the average of the values of the neighbouring pixels. Filters, other than the coordinates (u, v) to which they are applied, also have another important parameter: the *size*. The size of a filter defines how many neighbouring pixels will be used in the computation of the new pixel value and thus determines the spatial support of the filter, which will be obliviously centred at (u, v); common sizes are 3x3 or 5x5. In addition, also the *shape* of a filter is important. Filters can in fact be quadratic, rectangular or even circular depending on the results one wants to obtain.

3.2.1 Filter Masks

The basic concept behind filters is the *mask* $H(i, j)$ which specifies the weights each source pixel has in the definition of the new value. Filters can then be categorized in *linear*

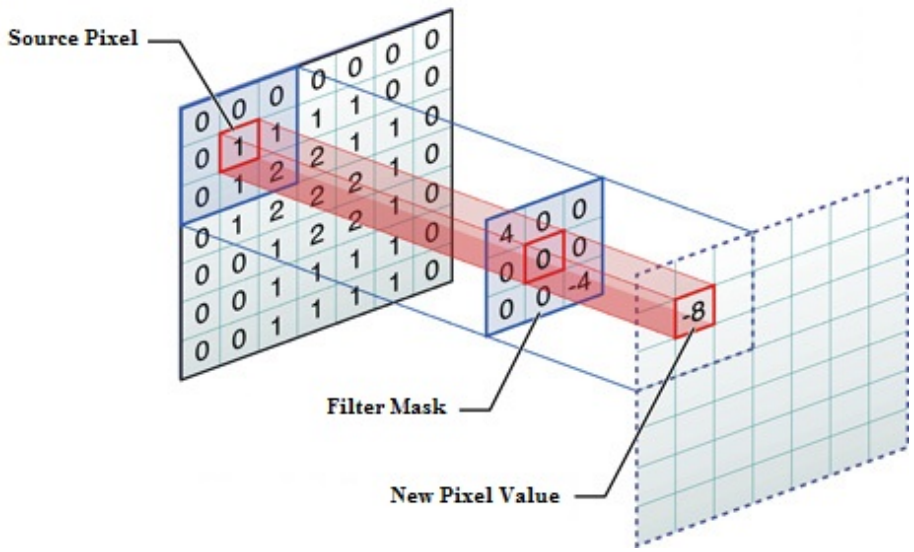


Figure 3: Mechanics of a filter: the filter mask (in the middle) is centred on every single source pixel to create a new, filtered image (identical in size to the source). The content of the mask and its mapping function vary depending on the filter applied.

and *non-linear*, depending whether those weighted pixels values are combined by a linear (*i.e.* a weighted sum) or non-linear function. The filter mask has its own coordinate system defined by the origin $H(0,0)$, usually positioned at the centre: negative values are therefore allowed. A schematic description of a filter, illustrating their general mechanics, can be seen in Figure 3.

A Linear Filter Example

To better describe how filters work, consider this linear filter example. As already said, the filter mask specifies the coefficient for every neighbouring pixel. For every image position (u, v) the following steps are performed:

1. The filter is superimposed to the image so that $I(u, v)$ coincides with $H(0, 0)$
2. The filter coefficients $H(i, j)$ are multiplied with the corresponding source pixel values $I(u + i, v + j)$ and the products added.

3. The result is stored at the same position but in the new, filtered image $I'(u, v)$

More formally:

$$I'(u,v) = \sum_{(i,j)\in R_H} I(u+i, v+j) \cdot H(i,j) \tag{5}$$

However, an issue arises with the image borders given that the filter mask goes outside the boundaries of the picture. No bullet-proof method exists to cope with this problem; the most common solution is simply to artificially extends the boundary pixels beyond the limit of the image.

3.3 Morphological Filters

A particular type of filters, especially designed for binary images, are called *morphological filters* because their aim is to alter the local structure of an image in a predictable way. For example, they are able to fill small holes and remove certain structures such as thin lines and dots. They are particularly useful to remove noise and/or uninteresting objects from the image, to only obtain the most important details. Their approach can be described as *erode and expand*.

Assume a binary image must be filtered to remove the smaller structures but keep the bigger ones intact. A morphological filter would approach such problem as described in Figure 4. Firstly, the boundary pixels (*i.e.* the foreground ones in contact with the background) are identified and removed. Thus, the objects are iteratively shrunk; in this way the smaller structures disappear. Secondly, the remaining structures are grown again by the same amount: eventually, the new picture will be only composed of the bigger objects, approximately returned to their original size.

The two most common morphological filters are indeed *dilation* and *erosion*. For both of them we must however define certain basic concepts.

Pixels Neighbourhood

The neighbourhood of pixels can be defined in two ways:

- **4-neighbourhood** ($\aleph_4$): the four pixels in contact to a given pixels in the horizontal and vertical positions only.
- **8-neighbourhood** ($\aleph_8$): the pixels in $\aleph_4$ plus the four contiguous pixels on the diagonals.

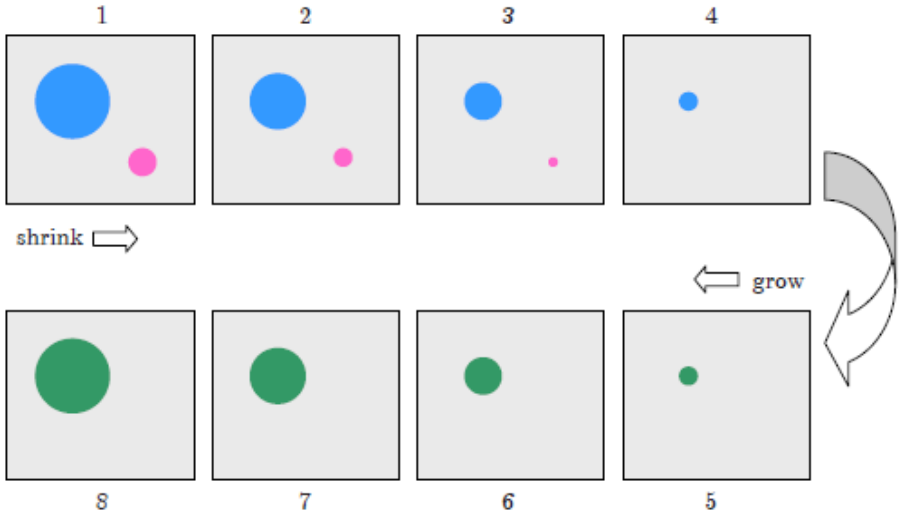


Figure 4: General mechanics of a morphological filter to remove small objects: first, the objects are shrunk (note that the small object disappeared) and subsequently grown again.

The structuring element

The structuring element is simply another name to define a filter mask. In morphology, such structuring elements only contain 0 and 1 values:

$$H(i,j) \in 0,1 \tag{6}$$

and the origin of $H(i,j)$, usually positioned at the centre, is called *hotspot*. As in normal filters, it denotes the origin of the coordinate system (*i.e.* the (0,0) vector).

3.3.1 Point Sets

In order to present the two most common morphological operations, it is useful to conceive binary images (and filters too) as sets of two-dimensional points. Indeed, given an image $I(u,v) \in (0,1)$, the set Q_I contains all the coordinate pairs $p = (u,v)$ of all foreground pixels:

$$Q_I = \{p \mid I(p) = 1\} \tag{7}$$

This notation allows to represent morphological filters as simple operations between point sets. For example, inverting an image $I \rightarrow \bar{I}$ (that is, exchanging foreground with

background) is equivalent to constructing the complement set:

$$Q_{\bar{I}} = \bar{Q}_I = \{p \ \in \ \mathbb{Z}^2 \ | p \ \notin \ Q_I\} \tag{8}$$

3.3.2 Dilation

Dilation is the morphological operation that, as described above, allow the image objects to grow. Formally, it can be expressed as:

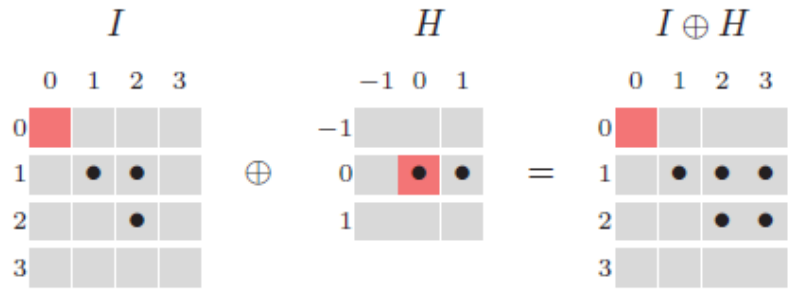
$$I \oplus H \equiv \{(p+q) \ | \ \forall p \in I, \ q \in H\} \tag{9}$$

The result is therefore the vector sum of all possible pairs of coordinates from the starting sets I and H. In a similar manner, dilation can also be seen as a replication of the filter mask H onto every foreground pixel in I:

$$I \oplus H \equiv \bigcup_{p \in I} H_p = \bigcup_{q \in H} I_q \tag{10}$$

where H_p and I_q denotes the sets H and I moved by, respectively, p and q. In Figure 5 it's possible to see a dilation filter in action[3]

[3]Important to note: in image processing, a different coordinate system is used. Firstly, the origin is usually in the upper-left corner of a matrix. Secondly, the axis are flipped in notation: in a pair of coordinates (u, v), u denotes the columns and v denotes the rows.



$$I \equiv \{(1,1),(2,1),(2,2)\},\ H \equiv \{(0,0),(1,0)\}$$

$$\begin{aligned} I \oplus H \equiv \{ & (1,1)+(0,0),\ (1,1)+(1,0), \\ & (2,1)+(0,0),\ (2,1)+(1,0), \\ & (2,2)+(0,0),\ (2,2)+(1,0) \} \end{aligned}$$

Figure 5: Dilation example. The image I is subject to the filter mask H which is replicated at every foreground pixel to generate $I \oplus H$. Note the coordinate system: in image processing, the axis are flipped and the first coordinate defines the columns, not the rows.

3.3.3 Erosion

Erosion works as the quasi-inverse of dilation. We could define it as:

$$I \ominus H \equiv \{(p+q) \mid \forall p \in I,\ q \in H\} \tag{11}$$

In other words, a point p is included in the result $I \ominus H$ if and only if the mask H is fully contained in the foreground pixels (when centred on p).

3.4 Image Regions

Finding *regions* (or *objects*) in an image is one fundamental step towards image classification: in fact our task is exactly to identify automatically what does the picture represent. The basic idea behind it lies in the definition of an image region: a group of contiguous foreground pixels. Many algorithm that allow for the identification of image regions exists and, obliviously, their job would be greatly simplified in the presence of binary images;

however, they all work according to a given single pattern: neighbouring pixels are iteratively grouped together to build regions in which all pixels within that region are given a single number (a *label*); this process is called *region labelling*.

Independently from which method we use, we would still need to identify a definition of pixel neighbourhood as seen above. Moreover, the pixels $I(u, v)$ will now have two values associated with them: the back/foreground identifier (either 0 or 1) and the region label:

$$I(u,v) = \begin{cases} 0 & \text{background pixel} \\ 1 & \text{foreground pixel} \\ 2, 3, ... & \text{region labels} \end{cases} \tag{12}$$

3.4.1 Flood Filling Labelling

As previously said, many different algorithms exists that allow for object recognition. Nevertheless, the most used (and practical)[4] method is called *flood filling*: it searches the entire picture for an unlabelled foreground pixel. One it has found it, it checks and *fills* also the neighbouring pixels. It proceeds sort of a flood (that's the reason of its name) throughout the whole region. There are three different ways in which this algorithm can be executed. They essentially differ in how the next pixel is selected:

- **Recursive Flood Filling**: as a recursive algorithm, it does not make use of data structures to store the pixels coordinates but uses local variables that are selected through the recursive calls to the procedure. Essentially, for each region a tree structure is constructed starting from the initial pixel and grown following the neighbouring structure of the region. However, given that the size of this recursion (and consequently the size of the required stack memory) is proportional to the size of the region, we incur the risk of exhausting the stack memory available. For this reason, recursive flood filling is not really used in practice but for tiny images.

- **Iterative Flood Filling**: also called ***depth-first***, it is simply a translation into an iterative algorithm of the recursive flood filling. It basically implements and manages its own *stacks* which store the adjacent but not yet visited pixels coordinates. By creating its own stack, the procedure leverage on the *heap* memory which is much larger, unconstraining the depth of the tree. It's called *depth-first* because the pixel

[4] See [2] section 2.1.1

tree is read by going all the way down of a branch to maximum depth before reading other branches[5]

- **Iterative Flood Filling (*breadth-first*)**: This procedure is actually identical as the depth-first version expect for the way the unvisited coordinates are stored: instead of using a stack, it uses a *queue*.

There exist another, much more complex region labelling algorithm called *Sequential region labelling*. However, it does not offer any real advantages over the iterative versions of flood filling. In practice, the most commonly used implementation is the *breadth-first* one, which allows for efficient performance even on complex and large images.

3.4.2 Region representation

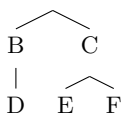
In order to understand some of the features presented in the next section, it is important to describe how to represent mathematically an image region. The most useful method is through *chain codes* also known as *Freeman codes*: starting from a given point x_s, we encode the region contour by the series of directional changes that occur to return to x_s. Formally: for a closed region $\mathscr{R}$ identified by the sequence of points $c_{\mathscr{R}} = [x_0, x_1, ..., x_{M-1}]$ where $x_i = (u_i, v_i)$, we can create the *chain sequence* $c'_{\mathscr{R}} = [c'_0, c'_1, ..., c'_{M-1}]$ by

$$c'_{\mathscr{R}} = \text{CODE}(\Delta u_i, \Delta v_i) \tag{13}$$

$$(\Delta u_i, \Delta v_i) = \begin{cases} (u_{i+1} - u_i, v_{i+1} - v_i) & \text{for } 0 \leq i < M-1 \\ (u_0 - u_i, v_0 - v_i) & \text{for } i = M-1 \end{cases} \tag{14}$$

and CODE$(\cdot)$ being the function used to represent directional changes according to the table below[6]:

Δu	1	1	0	-1	-1	-1	0	1
Δv	0	1	1	1	0	-1	-1	-1
CODE(Δu, Δv)	0	1	2	3	4	5	6	7

[5]A crude but efficient example: With *depth-first* the tree  would be read: A, B, D, C, E, F.

With *breadth-first* instead: A, B, C, D, E, F.

[6]constructed assuming the utilization of an 8-neighbourhood

It is possible to grasp the inner working of chain code sequences by looking at Figure 6

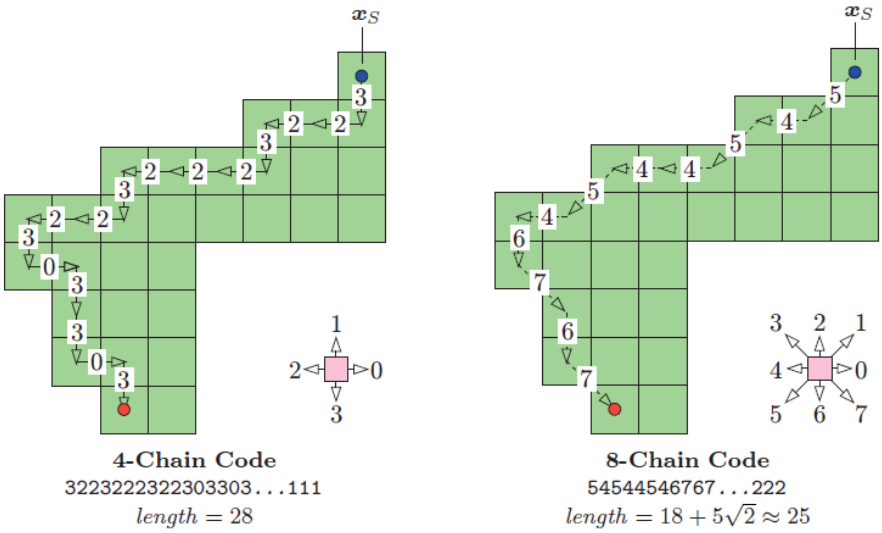


Figure 6: Defining a chain code sequence with both 4 and 8-neighbourhood. Starting from pixel x_s, we encode the direction of every movement following the contour of the region.

.

4 Feature extraction

Once we have identified and labelled the regions in an image we can start defining some geometrical characteristics about them. These characteristics are usually called *features*. In the context of image classification the extraction of features is of fundamental importance, because it defines the variables upon which to train the classifier. Those features must therefore incorporate enough information to allow the model to distinguish the different classes; at the same time though, we also need to minimize as possible the number of the selected features, to avoid computational issues in estimating the parameters of the model, as will be described in the next section.

Two of the simplest examples of features are the *area* and the *number of pixels* in a particular region. By calculating many different types of such region "qualities", one would be able to assemble a vector which describes the particular region in question. In the following section the most important features used in pattern recognition will be

presented, together with a mathematical explanation.

4.1 Geometrical Proprieties

A given region $\mathscr{R}$ could be defined as distribution, in two dimensions, of foreground pixels $x_i = (u_i, v_i)$ on the discrete set $\mathbb{Z}$:

$$\mathscr{R} = \{x_0, x_1, ..., x_{N-1}\} = \{(u_0, v_0), (u_1, v_1), ..., (u_{N-1}, v_{N-1})\} \tag{15}$$

4.1.1 Perimeter

Mathematically, the perimeter of a region $\mathscr{R}$ is simply defined as the length of its external boundaries (where $\mathscr{R}$ is connected). In the context of images, to calculate the perimeter we also need to take into account the definition of neighbourhood we are using: nevertheless, in either cases we will not get a perfect analytical answer. For example, when using the 8-pixel definition we must use some weighing depending on the direction of the segments: 1 for the horizontal and vertical ones, $\sqrt{2}$ for the diagonal ones. Given an 8-neighbourhood defined chain code $c'_{\mathscr{R}}$ the peremeter estimation would be:

$$\mathrm{P}\mathscr{R} = \sum_{i=0}^{M-1} \mathrm{length}(c'_i) \tag{16}$$

where:

$$\mathrm{length}(c) = \begin{cases} 1 & \text{for } c = 0, 2, 4, 6 \\ \sqrt{2} & \text{for } c = 1, 3, 5, 7 \end{cases} \tag{17}$$

4.1.2 Area

The area of $\mathscr{R}$ is quite simple to compute by simply counting the number of pixels of which is formed:

$$\mathrm{Area}(\mathscr{R}) = |\mathscr{R}| = \mathrm{N} \tag{18}$$

Another approach to estimate the area would be to make use of its own contour identified by the coordinate pairs $(x_0, x_1, ..., x_{M-1})$ and the Gaussian area formula for polygons:

$$\text{Area}(\mathscr{R}) \approx \frac{1}{2} \cdot \left| \sum_{i=0}^{M-1} (u_i \cdot v_{(i+1) \bmod M} - u_{(i+1) \bmod M} \cdot v_i) \right| \tag{19}$$

Simple geometrical properties such as area and perimeter are not influenced by rotation or translation of the objects/regions in the images. However they are for sure affected by scaling *i.e.* changes in size. Fortunately, among the countless number of features, there exist some that are invariant to both scale, rotation and translation.

4.2 Moments & Statistical Proprieties

Statistical proprieties arise from the simple translation of $\mathscr{R}$ as a set of points distributed in two-dimensional space. An important concept in this case is that of *centroid*.

4.2.1 Centroid

The centroid $\bar{\mathbf{x}} = (\bar{x}, \bar{y})$ of a binary, two-dimensional set of points is simply the arithmetic mean of the of the coordinates in the x and y directions.

$$\bar{x} = \frac{1}{|\mathscr{R}|} \cdot \sum_{(u,v) \in \mathscr{R}} u \tag{20}$$

$$\bar{y} = \frac{1}{|\mathscr{R}|} \cdot \sum_{(u,v) \in \mathscr{R}} v \tag{21}$$

4.2.2 Central Moments

The definition of centroid above is simply a special case of the more general concept of *central moments* of order p, q of a distribution. In particular, by shifting the centroid we are able to create translation invariant features:

$$\mu_{pq}(\mathscr{R}) = \sum_{(u,v) \in \mathscr{R}} I(u,v) \cdot (u - \bar{x})^p \cdot (v - \bar{y})^q \tag{22}$$

4.2.3 Normalized Central Moment

Central moments are actually influenced by the absolute size of the image region: indeed, their value is dependent upon the distance of all the points from the centroid $\bar{\mathbf{x}}$. To avoid

this issue, it is possible to uniformly *normalize* the region $\mathscr{R}$ by a factor $s \in \mathbb{R}$. This transformation will multiply the central moment by:

$$s^{p+q+2} \tag{23}$$

Hence, normalized moments can be derived as:

$$\bar{\mu}_{pq}(\mathscr{R}) = \mu_{pq}(\mathscr{R}) \cdot \left(\frac{1}{\mu_{00}(\mathscr{R})} \right)^{(p+q+2)/2} \tag{24}$$

The resulting feature is not only translation invariant, but now also scale invariant.

4.2.4 Hu Moments

As described thoughtfully in the work of [4], from the normalized moments $\bar{\mu}_{pq}(\mathscr{R})$ it is actually possible to define a third type of moments which are translation, scale and rotation invariant: hence very useful for pattern recognition. They are called *Hu moments* and are calculated as:

$$\begin{aligned}
H_1 &= \bar{\mu}_{20} + \bar{\mu}_{02} \\
H_2 &= (\bar{\mu}_{20} - \bar{\mu}_{02})^2 + 4\bar{\mu}_{11}^2 \\
H_3 &= (\bar{\mu}_{30} - 3\bar{\mu}_{12})^2 + (3\bar{\mu}_{21} - \bar{\mu}_{03})^2 \\
H_4 &= (\bar{\mu}_{30} + \bar{\mu}_{12})^2 + (\bar{\mu}_{21} + \bar{\mu}_{03})^2 \\
H_5 &= (\bar{\mu}_{30} + \bar{\mu}_{12})(\bar{\mu}_{30} - 3\bar{\mu}_{12}) \left[(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - 3(\bar{\mu}_{21} + \bar{\mu}_{03})^2 \right] \\
&+ (\bar{\mu}_{21} + \bar{\mu}_{03})(3\bar{\mu}_{21} - \bar{\mu}_{03}) \left[(3\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2 \right] \\
H_6 &= (\bar{\mu}_{20} - \bar{\mu}_{02}) \left[(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2 \right] + 4\bar{\mu}_{11}(\bar{\mu}_{30} + \bar{\mu}_{12})(\bar{\mu}_{21} + \bar{\mu}_{03}) \\
H_7 &= (3\bar{\mu}_{21} - \bar{\mu}_{03})(\bar{\mu}_{30} + \bar{\mu}_{12}) \left[(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - 3(\bar{\mu}_{21} + \bar{\mu}_{03})^2 \right] \\
&- (\bar{\mu}_{30} - 3\bar{\mu}_{12})(\bar{\mu}_{21} + \bar{\mu}_{03}) \left[(3\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2 \right]
\end{aligned}$$

In practical terms, they are usually log-transformed since they can take very large values, creating problems in model convergence.

4.2.5 Eccentricity

Eccentricity refers to how much a region deviates from being circular. An initial approach would be to rotate the region until we can fit such region in a bounding ellipse with

maximum aspect ratio. However, such process would be highly inefficient given that it would require many rotations. A much better approach can be obtained by using the region moments: not only this calculation requires significantly less computing power, it also provides a much more efficient and accurate estimate for the eccentricity.

Eccentricity can be formally defined as:

$$\mathrm{Ecc}(\mathscr{R}) = \frac{a_1}{a_2} = \frac{\mu_{20} + \mu_{02} + \sqrt{(\mu_{20} + \mu_{02})^2 + 4\mu_{11}^2}}{\mu_{20} + \mu_{02} - \sqrt{(\mu_{20} + \mu_{02})^2 + 4\mu_{11}^2}} \tag{25}$$

with $a_1 = 2\lambda_1$ and $a_2 = 2\lambda_2$ and λ_1, λ_2 are the first and second eigenvalues of the matrix of moments $\mathbf{A}$:

$$\begin{bmatrix} \mu_{20} & \mu_{11} \\ \mu_{11} & \mu_{02} \end{bmatrix}$$

that is, the matrix formed by the central moments of the region $\mathscr{R}$. The eccentricity can take values between $[1, \mathrm{inf})$ where $\mathrm{Ecc} = 1$ defines a perfect circle while $\mathrm{Ecc} > 1$ defines increasingly elongated shapes. As far as geometric properties, eccentricity is invariant of both scale and orientation. An important consideration is needed for the values λ_1 and λ_2. In fact they are connected to the proportions of the ellipse positioned on the centroid $(\bar{x}, \bar{y})$ and oriented in the same direction of the major axis of the region. Therefore, we can directly compute the length of the ellipse major and minor axis:

$$r_a = 2 \cdot \left(\frac{\lambda_1}{|\mathscr{R}|} \right)^{1/2} \tag{26}$$

$$r_b = 2 \cdot \left(\frac{\lambda_2}{|\mathscr{R}|} \right)^{1/2} \tag{27}$$

where $|\mathscr{R}|$ defines the number of pixels in the region. The ratio of those two axis is commonly used as possible feature for pattern recognition.

5 Neural Network

For this pattern recognition task, after having extracted the chosen features, a Neural Network (NN) classifier was used. The basic idea behind such a model is to create a linear transformation of the inputs and use it to model the output as a non-linear function. Such approach allows for a powerful learning tool, especially suited for pattern recognition from

visual data. The following section presents the basic theory of Neural Nets which will be used in the application. A more detailed introduction to this topic can be found at [5]. Nevertheless, it is important to note that many different types of neural nets exists, both supervised and unsupervised and with different mechanics/structures: a very well documented (even if old) source of information on all this variety can be found at the NN FAQ.

5.1 General Architecture of neural nets

NNs can be easily described as a multi-layer statistical model usually described using a *network diagram* such as the one in Figure 7. It represent a neural network with a single *hidden layer* architecture: the first layer, denoted with X_p, defines the input variables: in the pattern recognition application, the inputs can either be raw pixels or derived features. Next, the second layer, denoted by Z_M, represents the *hidden layer* created by a linear combination of the inputs. Finally, the last layer Y_K identifies the output(s), created by a function of linear combinations of the previous layer. Of course, in the Regression case, $K = 1$, while for Classification K would be equal to the number of classes, where the kth unit is the predicted probability (for that observation) of belonging to the kth class.

Formally, we could represent such network as (X defines the vector of inputs):

$$\begin{aligned} Z_m &= \sigma(\alpha_{0m} + \alpha_m^T X), m = 1, ..., M \\ T_k &= \beta_{0k} + \beta_k^T Z, k = 1, ..., K \\ f_k(X) &= Y_k = g_k(T), k = 1, ..., K \end{aligned}$$

Traditionally, NNs' coefficients are called *biases* (for the intercepts: α_{0m} and β_{0k}) and *weights* (for the slopes: $\alpha_m^T X$ and $\beta_k^T Z$). From the previous equations, the reader can clearly see that both the hidden layer Z_m and the output Y_k include a transform of the respective linear combinations: this is done to both allow a wider class of linear models and to "translate" the output in a statistically meaningful form. In particular, $\sigma(v)$ is denoted *activation function*: many different functional forms can be chosen at this stage. The most common one is the *sigmoid* (Figure 8) which maps the input v in (0, 1) and is defined as:

$$\sigma(v) = \frac{1}{(1 + e^{-v})} \tag{28}$$

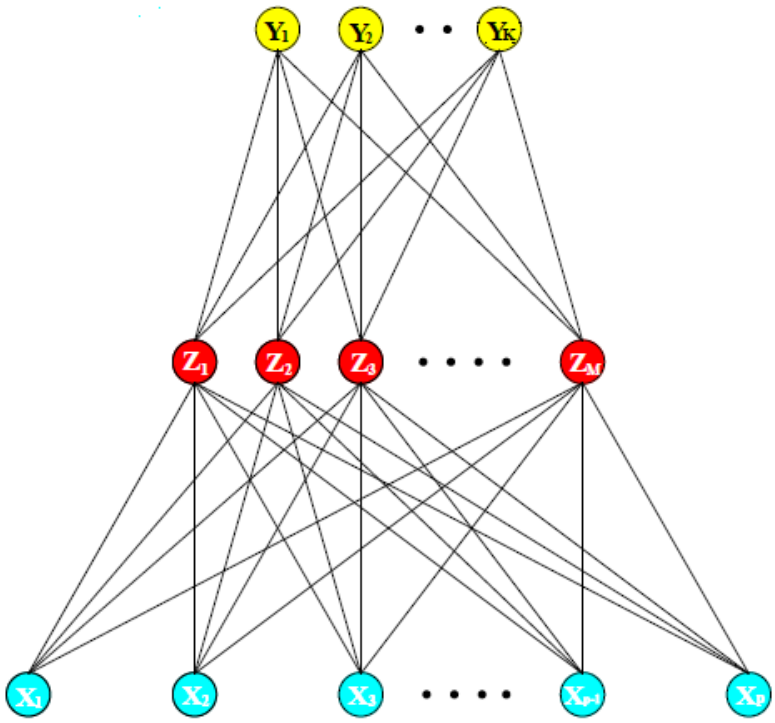


Figure 7: Network diagram of a single layer, feed-forward NN. X_p defines the inputs, Z_M the hidden units (or *neurons*) and Y_K the outputs. Credits to [6]

.

Notice that if $\|v\|$ is small $\sigma(v)$ can be approximated by a simple linear function, *i.e.* if the model parameters are small, v will be small and the function becomes approximately linear. As discussed in [7], we could hence say that a network with sigmoidal activation also contains a linear network as a special case (*i.e.* when the parameters are close to zero). Other common choices for activation functions are the *hyperbolic tangent* giving real values in (-1, 1), and the *identity*. The latter makes the whole neural network to simply become a linear model in the inputs.

At the same time, also $g_k(T)$ can be specified under different forms. In the regression case, the *identity* is usually chosen. On the contrary, in classification problems (as this case) the most used functions are the *logistic* and its multi-class generalisation, the *softmax*:

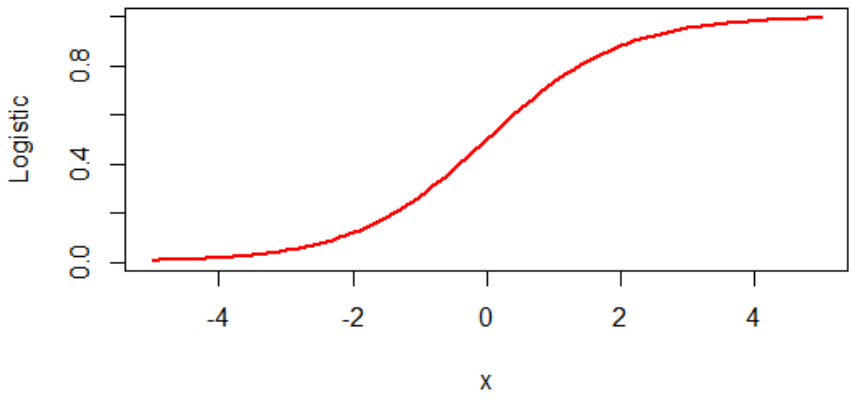


Figure 8: Plot of the sigmoidal activation function given by 28

.

$$g_k(T) = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}} \tag{29}$$

All the theory presented thus far refers to the so called *units* (or *neurons*) of NNs. Obliviously, the scientist must not only decide which functional form(s) to choose: he/she must also decide how many *hidden units* to estimate in how many *hidden layers*. In fact, multiple strata can be stacked above each other, to allow the creation of functional mappings of increasing complexity.

5.1.1 Connections

Neural networks have another important component: the *connections* between units. We can indeed identify three main architectures (see [8]):

1. Feed-forward NNs
2. Recurrent NNs
3. Convolutional NNs

Feed-forward NNs are the most common and the simplest type. In them, the information can only proceed in one direction: from the input layer, passing through the hidden layer(s) to the output nodes. No cycles or loops are allowed. On the contrary, *recurrent* neural networks are based on such cycles, allowing the state of the model at any time to be dependent on previous states. In other words, the output of some units are fed back either to the same unit or to units in preceding layers. See [9] for a detailed description of recurrent neural nets. Finally, *convolutional* neural nets occupy a particular position, since they are specifically suited for image processing. Moreover, their construction is not immediately intuitive: for this reasons, I shall discuss them in a detailed manner.

5.1.2 Convolutional neural networks

Ordinary feed-forward networks are not well suited to process raw images (*i.e.* where the inputs are directly the pixels) due to the scale that such approach would require. In fact, the big dimensionality of picture increases significantly the number of parameters to estimate. For example, with a picture of 200x200x3 (respectively: width, height and colour channels, also called depth) a single fully connected neuron in the first layer would have 200*200*3 = 120,000 parameters to estimate, which is obliviously problematic if we think we need to build multiple neurons in multiple layers.

For such reason, convolutional neural nets have been developed: their architecture is sensibly adjusted in order to reduce the number of parameters required. Their neurons are not fully-connected with the previous layer: on the contrary, each unit is connected only to a small subset of the stratum before. This characteristic is called *local connectivity* and is controlled by the *receptive field* or *filter*, an hyper-parameter which refers to the image subset each neuron is connected to. For example, a receptive field of dimensions 5x5, means that each neuron is modelling a different 5x5 image window (note that the depth dimension, *i.e.* colour, cannot be modified). Its mechanics is similar to the filters used in image processing.

Two other hyper-parameters are especially important for the definition of a convolutional layer. The first is called *depth*: it controls the number of filters trained to detect different features/pattern on the same picture spot. The second hyper-parameter is the *stride* which defines how many pixels should the receptive field slide each time. Both such parameters control for the size of the output that will be produced by such layer. Figure 9 depicts an example of a convolutional layer, in which the input picture is 32x32x3. The reader can clearly notice that there are five neurons modelling the same region in the

picture and at full colour channels.

Another fundamental difference of convolutional neural net is the so called *parameter sharing* which allows for a dramatic reduction of the number of weights. To enable this, we have to assume that if one feature is useful for prediction at one position (x_j, y_j), that same feature will be also useful at another position (x_i, y_i), with $j \neq i$. In other words, if we define a single depth dimension as a *depth slice* (*e.g* a 100x100x3 image has three 100x100 depth slices), we are going to constrain all units along each depth slice to use the same parameters, allowing for a drastic reduction of model complexity. For a more detailed review of convolutional neural nets, see [10].

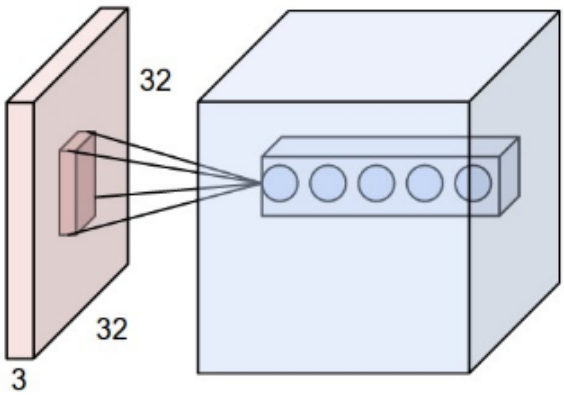


Figure 9: An example of convolutional layer. The input has size 32X32 with 3 colour channels. Notice the five neurons all looking at the same receptive field. Each of them will model different features of that particular spot.

.

5.1.3 Architectural choices

As the reader can appreciate, neural network construction is clearly not an easy process. While other statistical models only require the tuning of a handful of parameters, usually through generalisation error, neural nets pose a significant challenge and their design process cannot, at current state of understanding, be automatized. The scientist is left with several decisions to make, where each one of them could potentially affect the performance of the model:

- The number of hidden layers

- How many units/neurons for each layer
- Which type of connection to implement
- How to connect the units/layers: it is in fact possible to skip a layer or fully connect all units.
- Which activation function to use
- Which output function to implement

For the absence of clear-cut architectural rules, and the overwhelming number of decision to be made, neural network construction is sometimes referred as an "art".

5.1.4 A neural net example

To summarize (and clarify) how a neural network could be defined, I provide an architectural example. In this simple case, a two levels classification problem must be addressed. A possible model could be:

$$\begin{aligned}
\ln\left(\frac{\hat{p}}{(1-\hat{p})}\right) &= \hat{\beta}_0 + \hat{\beta}_1 z_1 + \hat{\beta}_2 z_2 + \hat{\beta}_3 z_3 && \text{Output Layer} \\
z_1 &= \tanh(\hat{\alpha}_4 + \hat{\alpha}_5 x_1 + \hat{\alpha}_6 x_2) && \text{Hidden neuron one} \\
z_2 &= \tanh(\hat{\alpha}_7 + \hat{\alpha}_8 x_1 + \hat{\alpha}_9 x_2) && \text{Hidden neuron two} \\
z_3 &= \tanh(\hat{\alpha}_{10} + \hat{\alpha}_{11} x_1 + \hat{\alpha}_{12} x_2) && \text{Hidden neuron three}
\end{aligned}$$

In such case, we have a logit output function and hyperbolic activations. We can clearly see that the two inputs x_1 and x_2 are fully connected in a forward way to the three neurons z_1, z_2 and z_3. Hence, a single hidden layer exists. Those three neurons are then fed to the output layer, which combines them to produce the logit.

5.2 Fitting procedure for neural networks

Fitting neural nets entails optimizing the chosen error function with respect to the set of parameters called θ. Nevertheless, those models, due to their construction, pose a particular problem called *credit assignment problem* (see [7]). If the output of a unit is incorrect when

the neural network is presented with an input vector there is no way of determining which of the hidden neurons should be held responsible for the error: therefore it is impossible to define which particular weight to adjust. Fortunately, the solution to this problem exists. If the considered network has a differentiable activation function, then the activations of the output neurons become themselves differentiable functions of both the inputs and θ. Consequently, if we also define an error function differentiable with respect to the outputs, then this error itself is differentiable in the weights. Hence, we are now able to differentiate the error function with respect to the weights and use such derivatives to optimize for θ that minimizes the error. The procedure is usually executed with *gradient descent* or *quasi-Newton's method*. It is therefore important to underline the two-step procedure at work: in the first phase the derivatives of the error function with respect to θ are evaluated using an algorithm called *back-propagation* (popularized by [11]). Then, in the second phase, we use such derivatives to find the parameters that minimize the error function: such process is repeated multiple times, with each pass called an *epoch*. Note that the last step can be tackled using a variety of optimization procedures[7].

5.2.1 Back-propagation of the errors

As defined previously, θ denotes the full set of weights or parameters. More formally:

$$\begin{aligned} \{\alpha_{0m}, \alpha_m; m = 1, ..., M\}, \quad & M(p+1) \quad \text{weights} \\ \{\beta_{0k}, \beta_k; k = 1, ..., K\}, \quad & K(M+1) \quad \text{weights} \end{aligned}$$

First of all, to fit a neural network (similarly to any other statistical classifier/regressor) an error function must be chosen. The most common options for are the *Residual sum of squares* for the regressors:

$$R(\theta) = \sum_{k=1}^{K} \sum_{i=1}^{N} (y_{ik} - f_k(x_i))^2 \tag{30}$$

Or the *cross-entropy*:

$$R(\theta) = -\sum_{i=1}^{N} \sum_{i=1}^{K} y_{ik} \log f_k(x_i) \tag{31}$$

[7]Such procedures are beyond the scope of this document. The interested reader is referred to [12]

After having chosen the $R(\theta)$, it is possible to proceed with back-propagation. Here, a general analytical solution using gradient descent will be provided:

The derivatives of $R(\theta)$ with respect to the parameters are:

$$\frac{\partial R_i}{\partial \beta_{km}} \tag{32a}$$

$$\frac{\partial R_i}{\partial \alpha_{ml}} \tag{32b}$$

Subsequently, a gradient descent update at the iteration $r+1$ is (where γ is the *learning rate* discussed later):

$$\begin{aligned} \beta_{km}^{(r+1)} &= \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \beta_{km}^{(r)}} \\ \alpha_{ml}^{(r+1)} &= \alpha_{ml}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \alpha_{ml}^{(r)}} \end{aligned} \tag{33}$$

we can now define formula 32 as:

$$\frac{\partial R_i}{\partial \beta_{km}^{(r)}} = \delta_{ki} z_{mi} \tag{34}$$

$$\frac{\partial R_i}{\partial \alpha_{ml}^{(r)}} = s_{mi} x_{il} \tag{35}$$

δ_{ki} and s_{mi} are called errors at the output and hidden layer respectively. In particular:

$$s_{mi} = \sigma'(\alpha_m^T x_i) \sum_{k=1}^{K} \beta_{km} \delta_{ki} \tag{36}$$

which is called the *back-propagation equation*. The whole algorithm is initialized with some weights (usually from a random uniform distribution), which allow a first estimate of the output errors δ_{ki}. Then, such errors are back-propagated via formula 36 to find s_{mi}. Finally, both δ_{ki} and s_{mi} are used to compute the gradients in 34 and 35.

Back-propagation is advantageous thanks to its local effect: each hidden neuron passes and receives the errors only with the neurons connected to it. This allows for easy parallelization when big problems require more computing power. We also referred to a parameter γ, calling it the *learning-rate*. Its role is to control how rapidly we fluctuate the parameters at each *epoch* (*i.e.* each single iteration r): a large γ leads to a faster optimization but it's highly likely to return sub-optimal parameter estimates.

5.3 Avoiding over-fit

Neural networks often present over-fitting, due to the already specified problems in choosing the right architecture: the problem may arise from too many weights and/or layers. An over-fit to the data creates a significant problem, since our model will not only model the signal but also the noise of our sample, significantly deteriorating its generalisation performance on the whole population. This behaviour is clearly summarized in Figure 10: the prediction error shows two different behaviours as we increase complexity (*i.e.* the number of parameters). It will continue to decrease and eventually reach the minimum if estimated on the training sample, while it will first decrease and then rise again if measured using a test sample. The objective is to create a model with an optimal level of complexity, that is where the minimum test/generalisation error is.

To reach such optimal point, two approaches can be devised. The first is based on using a separate test sample on which to estimate the generalisation error. In this case the user will stop fitting the neural network when such estimate begins to deteriorate (*i.e.* increases). The second approach, called *weight decay*, is much more explicit: it consists of including a *penalty term* to the error function. Hence, our objective function to minimize becomes:

$$\tilde{R}(\theta) = R(\theta) + \lambda J(\theta) \tag{37}$$

where:

- $R(\theta)$ is the chosen error function.
- $\lambda \geq 0$ is a tuning parameter: the greater its value, the greater the penalty and the more linear the neural network.
- $J(\theta)$ is a penalty term, a function of the parameters θ.

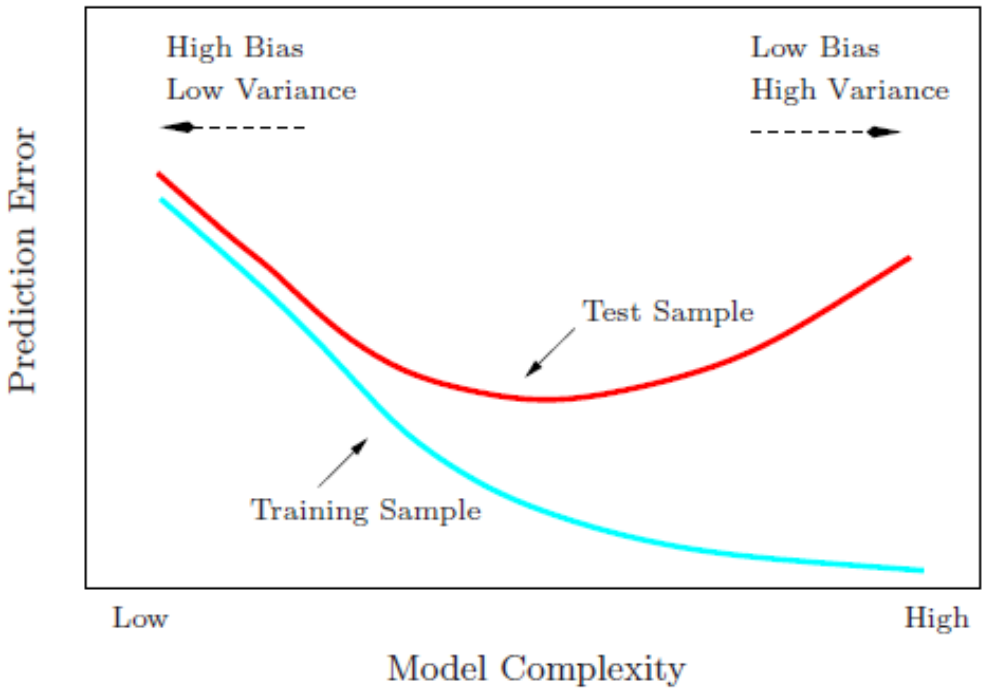


Figure 10: Training and test/generalisation error as a function of model complexity. Credits to [6]

The penalty term $J(\theta)$ can be defined in many forms. The most common one is the following. Its effect is simply to add $2\beta_{km}$ and $2\alpha_{ml}$ to the respective gradients in formula 33:

$$J(\theta) = \sum_{km} \beta_{km}^2 + \sum_{ml} \alpha_{ml}^2 \tag{38}$$

Obliviously, the optimal value for λ must be found using the estimates for generalisation error. The most common method is through Cross-Validation.

5.4 Issues of Neural Networks

As stated before, constructing and fitting a Neural Net is more of an art: no clear cut rules exists. In addition, the created model is highly likely to be overparametrized and the optimization surface will be non-convex and unstable, that is significantly influenced by the

starting values of the parameters. It is possible to identify four main problems that occur with neural net which are described below. In addition, such models are also characterized by an obvious lack of interpretability. Nevertheless, they are very useful in such context where the ratio of signal-to-noise is high.

5.4.1 Values for initialization

Choosing the starting values is of paramount importance when fitting such models. As stated before, when the parameters are close to zero, the operative part of the sigmoid (if we are using a sigmoidal activation function) is roughly linear. Commonly, starting weights are taken to be random numbers near zero, making the initial model linear. At subsequent epochs (*i.e.* each back-propagation pass), the units introduce non-linearities where needed. Moreover, starting with parameters to be exactly zero leads to zero derivatives and perfect symmetry, making the model impossible to evolve. On the contrary, starting with very large values usually leads to degenerate solutions.

5.4.2 Scale of the inputs

The quality of the final solution can be significantly affected by the scale of the inputs: this because scaling the inputs determines the scaling of the parameters in the output layer. Hence it is usually good practice to standardize the inputs to have mean zero and standard deviation one, allowing the user to to choose a reasonable range for the random starting parameters.

5.4.3 Architectural choices

As said before, neural networks construction has an astonishingly wide array of details to be arranged. In general though, as a rule of thumb, it is better to have too many hidden units than too few. In fact, if not enough hidden units are present, the model will not be flexible enough and will be unable to capture non-linearities. On the contrary, with too many hidden units the model will probably overparametrized but we can still use weight regularisation/decay to control for complexity. With respect to the number of hidden units, it usually reasonable to start with a large number (large in relation to the number of inputs). On the contrary, the number of hidden layers is largely guided by experiments: each layer extracts different features of the input. Hence, if we use multiple layers, we will be able to construct hierarchical features at differing levels of resolution.

5.4.4 Multiple minima

The issue of multiple minima is largely connected to the choice of starting values. The error function $R(\theta)$ is non-convex and hence characterized by multiple local minima. This creates a problem, which can however be (partly) solved by trying multiple starting values and choosing the one giving lowest generalisation error. A better approach could be, as suggested by [13], to average the predictions of all those different models. This last method is largely different from averaging the parameters, since the non-linearities will generate a very poor solution. Lastly, one could also try with *bagging*, by averaging the predictions from models trained using randomly perturbed data.

6 Application

All the theory discussed so far has been used to construct a pattern recognition algorithm; in other words the objective of the task was to create a specific program able to classify images according to which object they were representing. Specifically, the pictures represented 120 different species of plankton: classifying such pictures in the correct species was the aim of this application. The whole project has been conducted used the Python language [14] version 3.5.2. The CPU used was an Intel(R) i7-2670 QM at 2.20 GHz running on with Windows 7 OS.

6.1 The Data

The data was provided by Hatfield Marine Science C.enter of the Oregon State University. Of 74.1 Mb in size, it comprised 30k grey-scale images (of various pixel sizes) of planktons. They were divided into 121 different species (*i.e.* classes), each class presenting varying number of observations. Every raw image was already passed through a region-recognition program, meaning that each image contained only a single plankton specimen, with some random noise such as decomposing organic material (commonly known as *whale snot*). The specimens could be oriented in any direction in 3D space, hence different orientations were already taken into account naturally. Finally, some of the images where significantly noisy and ambigous making even manual identification impossible: for such instances, an Unknown category was present.

6.1.1 Collection

The images were collected with a specialized tool called Situ Ichtyoplankton Imaging System (ISIIS), which use line shadowgraph imagery and a line-scan camera. The continuous image produced is 2048 x 2048 pixel frames from which the region of interest were extracted: those regions are exactly the images provided in the dataset. The initial classification into different species was carried out manually: an interesting detail is that even a trained team cannot achieve a 100% accuracy: reasonable estimates lie between 84% to 95% as argued by [15].

6.1.2 Exempla

Plankton is defined as a diverse ensemble of organisms that live in water but that cannot move against current. Such organisms vary significantly from one another, from algae to jellyfishes to snails. Obliviously, given this diverse family, also their shapes vary significantly. This is very helpful for our image classification tasks allowing for a lower error. As an example see figure 11, which depicts a sample of highly dissimilar specimens.

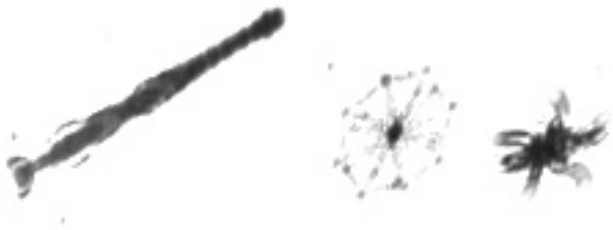


Figure 11: Example of very different object shapes; right *chaetognath other*, centre *protist other*, left *larva seastar bipinnaria*

.

The classes in figure 11 are in fact always well recognized by the models (see later sections). However, the majority of plankton species present in the sample are everything but well recognizable. See figure 12: the three specimens come from three different classes even though they are very similar, even for human standard[8]. In fact, we will see in the

[8]Their similarity is probably due to the fact that they come from the same group, namely the *acantharia*

following sections that all the models correctly recognize the last class while they do not achieve good performance on the other two. This is probably due also to the small sample size of the two misclassified categories: 13 and 17 respectively, against 889 for the last class.

Figure 12: Example of similar object shapes; right *acantharia protist big center*, centre *acantharia protist halo*, left *acantharia protist*

.

A similar impact of sample size can also be seen on another "family" of plankton, the *appendicularian* (see figure 13). Indeed, the first two classes, having a sample size of 696 and 532 are correctly recognized by the models while the last, with a sample size of only 16, fails to be classified.

---

protist

Figure 13: Example of similar object shapes; right *appendicularian s-shape*, centre *appendicularian slight curve* , left *appendicularian fritillaridae*

.

6.2 Extracting Features

As already specified in previous sections, the extraction of features is of paramount importance in the context of pattern recognition. The distinguishing characteristics must both incorporate as much information as possible in the smallest matrix dimensionality possible. Of course, we must have the same number of features for each observation.

In order to extract those features, some preprocessing was required. The following steps were ececuted:

1. **Thresholding**: given the raw grey-scale images some noise reduction was needed. The approach followed was to use binary thresholding (*i.e.* getting b/w images) with threshold value at the mean. This point operation could be therefore described as:

$$f_{threshold}(\alpha) = \begin{cases} 0, & \text{if } \alpha < \bar{\alpha} \\ 1, & \text{if } \alpha \geq \bar{\alpha} \end{cases} \tag{39}$$

where $\bar{\alpha}$ is the grand mean of all pixel values in the image:

$$\bar{\alpha} = \frac{\sum_{i=0}^{U} \sum_{j=0}^{V} \alpha_{i,j}}{U \cdot V} \tag{40}$$

with U as the number of columns and V as the number of rows. Thus, the images where transformed into binary.

2. **Dilation**: subsequently, smaller and uninteresting regions needed to be removed. This passage was required to ease the next step of region labelling. To achieve this, a dilation 4x4 filter $H(i,j)$ was used. Specifically, the algorithm implemented increased the value of pixel $I(i,j)$ to the maximum of all the pixels values found in the neighbourhood $H(i,j)$ of (i,j).

$$I'(i,j) = \max[H(i,j)] \tag{41}$$

3. **Region Labelling**: Once dilated, the regions would have to be recognized and labelled appropriately: thus, a proper flood-filling algorithm was used: the specific of its implementation can be found in [16].

It is possible to see the intermediate steps of image processing in Figure 14.

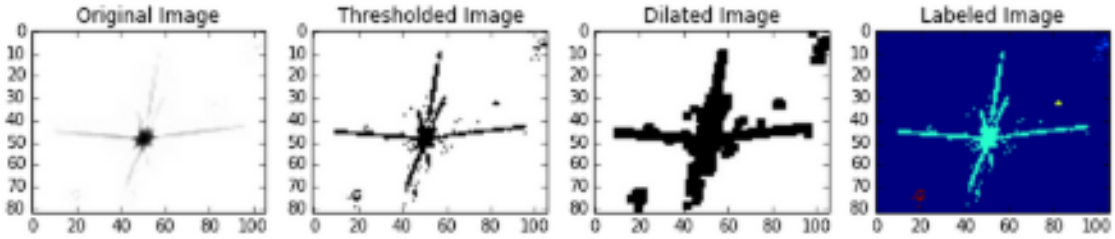


Figure 14: Examples of all processing steps and their effect on the images. From left to right: raw grey-scale image, binary thresholded image, image with dilated objects and properly labelled regions

.

Finally, only the largest region was considered under the assumption that it would represent the object of interest in every image, *i.e.* the plankton would be the biggest object present in all images. From such region, four features were extracted: all seven *Hu moments*, the *eccentricity* of the enclosing ellipse, the *area* or the number of pixels and the ratio between the *major* and the *minor* axes.

6.2.1 An Experimental Feature

In addition, an experimental feature was also added to the variable matrix. Specifically, every picture would be obliviously "closer" (or more similar) with its own class than with

other different species. Following this simple principle, the mean image for each class was computed from the raw grey-scale pictures:

$$\bar{I}^k(i,j) = \frac{\sum_{i=1}^{J} I_i^k(u,v)}{J} \tag{42}$$

where k defines the class/species of plankton and J the total number of images in that given class. Thanks to this class mean, we would therefore have at our disposal a sort of "representative" class picture. Subsequently, the image matrix was transformed into a vector in order to allow for the calculation of the *Euclidean distance* between every class k image I_i^k and every class mean $\bar{I}^k$. Thus, for every single image, a vector of Euclidean distances was obtained representing that image dissimilarity with respect to every single class mean: in practical terms, having 121 categories, every image I_i^k would have 121 distances *i.e.* 121 new features.

Image class means are not able however to retain fine details of the plankton specie. Nevertheless, they still vary significantly from one another, allowing for comparison between themselves. Figures from 15 to 17 are a sample of such means, together with a specimen of their original class.

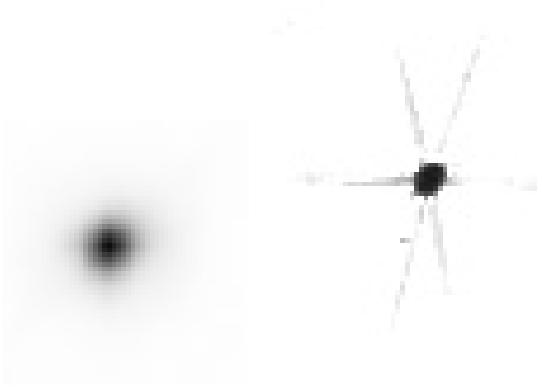


Figure 15: Left: the *acantharia protist* class mean. Right: example specimen of *acantharia protist*

.

Figure 16: Left: the *chaetognath other* class mean. Right: example specimen of *chaetognath other*

.

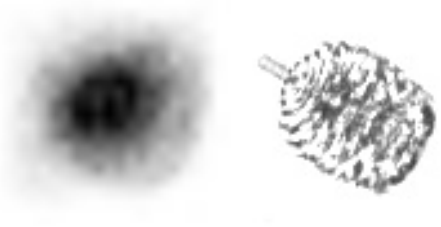


Figure 17: Left: the *copepod cyclopoid copilia* class mean. Right: example specimen of *copepod cyclopoid copilia*

.

6.3 Features performance

As said before, the best feature should condense as much information as possible for each class, and such information should be radically different for each one. A way to visualize if two classes are easily differentiated by mean of a feature is to look at their compared distribution *i.e.* how the two species are different in terms of a given feature. In this case it would be obliviously cumbersome to compare all classes in pairs of two. Nevertheless, it is still useful to pick some classes as examples and see how they perform. In the appendix A.1, it is possible to appreciate the performance of every feature used in this application given 18 species of plankton. For the features which contain multiple values, such as the seven *Hu moments* and the *mean euclidean distance*, only their first value is provided. All of them

perform quite satisfactorily, well separating some categories but being useless for others. For example, the *trichodesmium puff* and *chaetegnath other* are significantly different for all features[9] expect for area, which does not provide any useful information. We can see in fact that the geometrical properties are different for the two specimens as in figure 18: this is indeed reflected in most of the features. On the contrary, *area* does not do a good job (not even with other classes actually) because it is not scale invariant: each plankton area value depends on how close the image collection tool was to the specimen, creating a noisy feature that does not allow for object recognition. Therefore, the fundamental lesson is that few features are not enough for a classification problem, but it is always sensible to extract a large number of them, to allow more room for the classifier.

Figure 18: Left: an example of chaetegnath other. Right: an example of trichodesmium puff. Their geometric difference is remarkable.

.

6.4 Classification

After having constructed the feature vector for each image, the next step was to construct (and fit) a classifier. For this application, a neural network was chosen. However, as already seen in the neural net section, this type of models present inherent problems and

[9] even though the mean euclidean distance has some problems in separating them

ambiguities in their architecture. Therefore, multiple structures have been trialled. In the following all those different approaches will be described and discussed.

6.4.1 Performance Metrics

In order to evaluate and compare all the models against themselves, a measure of their performance is needed. Specifically, one would like to know how the classifier is doing in labelling the unseen inputs. For this reason, two measures have been chosen; they are defined for each category:

$$\text{Precision} = \frac{\text{tp}}{\text{tp} + \text{fp}} \tag{43}$$

$$\text{Recall} = \text{Sensitivity} = \frac{\text{tp}}{\text{tp} + \text{fn}} \tag{44}$$

where the numerator and denominator are found according to the following confusion matrix:

		Prediction	
		1	**0**
Truth	**1**	tp	fn
	0	fp	tn

However, given the big number of classes, comparing different models on how they perform on each class is not be feasible. Instead, a summary metric is needed. In this case, the measure of choice is the *multi-class Log-loss* defined as:

$$F = -\frac{1}{J} \sum_{i=1}^{J} \sum_{j=1}^{K} y_{ij} \cdot ln(p_{ij}) \tag{45}$$

with J as the total number of instances/pictures in a given class K, y_{ij} a binary variable indicating whether image I_i is truly in class j and p_{ij} is the model prediction for that image: in other words the predicted probability of image I_i being in class j. A perfect classifier score would reach 0.

6.4.2 Model architectures

Six models have been constructed. They all vary for many different details, from the input features used, to the layers and optimization iterations. The following is a schematic

description of their architectures. The following section will comment on their performance. The multi-class log-loss metric was obtained using 3-fold CV: a higher number of folds would have significantly increased the computational time.

Model one:

- **Features**: Euclidean distance between class means, minor/major axis ratio.
- **Layers and Units**: Just one layer with 150 hidden units. As activation function, the hyperbolic tangent was used.
- **Optimization iterations**: 3 gradient descent iterations with 0.01 of learning rate
- **Output function**: Softmax
- **Fitting time and Performance**: 5194 seconds and a log-loss of 3.1565.

Model two:

- **Features**: Euclidean distance between class means, minor/major axis ratio.
- **Layers and Units**: Two layers: first with 180 units, the second with 150. As activation function, the hyperbolic tangent was used.
- **Optimization iterations**: 3 gradient descent iterations with 0.01 of learning rate
- **Output function**: Softmax
- **Fitting time and Performance**: 6008 seconds and a log-loss of 3.3238.

Model three:

- **Features**: Euclidean distance between class means, minor/major axis ratio
- **Layers and Units**: Just one layer with 150 hidden units. As activation function, the hyperbolic tangent was used.
- **Optimization iterations**: 20 gradient descent iterations with 0.01 of learning rate
- **Output function**: Softmax

- **Fitting time and Performance**: 27492 seconds and a log-loss of 2.6282.

Model four:

- **Features**: Euclidean distance between class means, minor/major axis ratio, Hu moments, eccentricity and area.
- **Layers and Units**: One hidden layer with 200 hidden units. As activation function, the hyperbolic tangent was used.
- **Optimization iterations**: 20 gradient descent iterations with 0.01 of learning rate
- **Output function**: Softmax
- **Fitting time and Performance**: 39583 seconds and a log-loss of 4.2169.

Model five:

- **Features**: Euclidean distance between class means, minor/major axis ratio, Hu moments and eccentricity.
- **Layers and Units**: Two hidden layers: the first with 180 units, the second with 150. As activation function, the hyperbolic tangent was used.
- **Optimization iterations**: 20 gradient descent iterations with 0.01 of learning rate.
- **Output function**: Softmax.
- **Fitting time and Performance**: 42506 seconds and a log-loss of 2.4934.

Model six:

- **Features**: 48x48 raw images.
- **Layers and Units**: One convoluted hidden layer with hyperbolic tangent as activation function.
- **Receptive filter and Depth**: Square 3x3, 6 channels of depth.
- **Optimization iterations**: 10 gradient descent iterations with 0.01 of learning rate.
- **Output function**: Softmax.
- **Fitting time and Performance**: Not converged.

6.4.3 Model performances

For all the models, the single class performance metrics are available in the Appendix A.3. The first model provided the benchmark against which to assess all the other architectures. Its structure is pretty simple and obliviously could be significantly improved. In particular, the features used are just two. Moreover, the small number of optimization iterations reduced the computational time. Nevertheless, the risk of incurring in local minima was significantly high.

The single class performance of this model was everything but great. Even if a handful of classes achieved a precision of 0.5-0.6, the great majority of them were not classified correctly at all having a precision of 0.

The second model attempted to increase the complexity by adding a layer. As we can clearly see, adding a second layer actually decreased performance. Obliviously, there is always the risk that such result has simply been obtained due to a local minima, given the very low number of gradient descent iterations. Nevertheless, even though the multi-class log-loss was only 0.1673 higher, the single class performance was significantly worse: the almost totality of classes are not at all predicted correctly while the ones that are only achieve 0.10 to 0.2.

Therefore, it seems that with this combination of features, increasing the number of layers did not improve performance. On the contrary, over-fitting was highly likely.

Hence, to avoid the local minima problem, the third model was fitted with more iterations. Specifically, the model three structure is exactly the same as the first one. The only difference is the number of gradient descent iterations which had been increased from 3 to 20. Specifically, this model was fitted to understand the role of local minima in affecting the classification performance. As we can clearly see, the log-loss for model three was significantly lower than model one, suggesting that the latter has incurred in a local minima. Nevertheless, the fitting time increased considerably: GPU computing would for sure become handy at this stage. Obliviously, as the lower log-loss suggests, the single class performance was significantly improved, correctly predicting many more classes.

Next, in model four, an increased number of input features was tried. Nevertheless, augmenting the number of features while leaving the architecture virtually unchanged actually decreased performance. However, this result could have also been caused by the bigger number of hidden units: this could have potentially bring over-fitting. Of notable importance for this model is the single class performance, which is literally awful: just one

class is correctly predicted achieving a precision of 0.33. All other plankton species are stuck to 0. It would be unwise to blame local minima for this given the high number of iterations. Indeed, after having checked some summary statistics for the features vectors, the *area* was discovered to be practically homogeneous throughout most observations (see figure 20) in the Appedix), *i.e.* most regions of interest in the images had approximately the same number of pixels. This homogeneity in one of the variables could have been the source of the model bad performance.

Therefore, in model five *area* was removed form the feature matrix given that it was redundant by not providing enough information to differentiate between the classes. By far, this model had the best performance among all the ones constructed. In particular, even though the multiple log-loss was just 0.1348 lower with respect to the second-best architecture (model three), the single class prediction was much better: now the precision of the great majority of classes was actually higher than 0, with peaks of 0.6-0.72 for some of them. For this reason, this last model should be greatly preferred, even if the time to convergence is significantly bigger.

Finally, model six was attempted in order to test the performance of a convolutional network. Unfortunately, the convergence of such a model was stopped after several hours of fitting: therefore no results or comments can be expressed on its performance. For sure though, the time required to fit such architecture would not be negligible.

6.5 Kaggle note and winning model

The plankton classification task was also present as a Kaggle competition. The best model obtained in this application, that is model five, would have obtained position 579 out of 1049 participants. However, the model winning the competition was able to obtain a log-loss of only 0.5599. The winning team uploaded a GitHub page where they provide the code and explain their strategy.

To achieve such optimal performance they constructed an ensemble comprised of 40 different neural networks, most of which were convnets with number of layers between 6 and 16, with up to 27 million parameters. Little pre processing was done: they only rescaled the images to make the size homogeneous and they increased the sample size through random transforms:

- rotation: random with angle between 0 and 360 (uniform distribution)
- translation: random with shift between 10 and 10 pixels (uniform distribution)

- rescaling: random with scale factor between 1/1.6 and 1.6 (loguniform distribution)
- flipping: yes or no (bernoulli distribution)
- shearing: random with angle between 20 and 20 (uniform distribution)
- stretching: random with stretch factor between 1/1.3 and 1.3 (loguniform distribution)

Most of the convnets exploited what they called *cyclic pooling*, *i.e.* leverage on the rotational symmetry of the images to share parameter and limit the computational burden. In practice, each image occured 4 times in 4 different orientations. Each one is processed in parallel by the network and then the feature maps are pooled together. Figure 19 depicts the process.

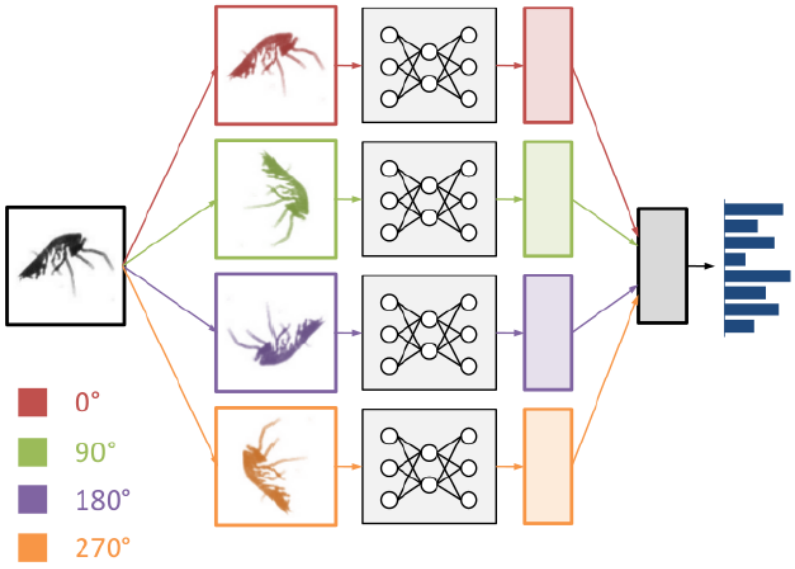


Figure 19: *Cyclic pooling* in the convnets.

.

In addition, they also built feature-based models to adjust the predictions of convnets. The selected features were:

- Image size in pixels
- Image moments to approximate for the size and shape

- Haralick texture features (see [17])

Interestingly, such feature-based networks were actually fused together with the convents immediately before the final layerm as if they were a totally new input. Important to note, is the very high number of gradient descent iterations for the convenets: 215,000. As noted in the model built in this application, such high number of steps allows to avoid local optima and significantly improves model performance. Apparently, using the GPU for fitting, their models took between 24 to 48 hours to converge.

6.6 Towards a better classifier

The performance of the six different models is anything but satisfying: for a real-life use, such models would be of little help, misclassifying most of the cases. Thus, a better classifier is needed. But how to reach this goal? The answer is provided by the Kaggle competition winners: increase complexity[10], *i.e.* fit the model using a higher number of inputs and layersunits. Indeed, as we have seen before, most of the classes are actually quite similar and it is obliviously difficult for a model to discriminate between them by simply using a handful of features. This because the geometric properties of those objects cannot be summarized in few metrics: the only way of extracting all the necessary information is to use the whole images, pixel by pixel. Such approach however does not require any particularly complex statistical tuning, but only sheer computing power. In fact, in order to process the entire images, a (or multiple) convnet is required; even though convnets share parameters, the problem still remain quite computationally intensive: we remind the reader that the convenet architecture attempted in model six did not converge in a reasonable amount of time and had to be stopped[11]. Using a "traditional" feed-forward neural net would be unwise for obvious reasons.

A second lesson that can be used to build a better classifier is to increase the number of optimization iterations. Indeed, in the six models trained, increasing that number always resulted in improved performance. Also in this case, it does not seem to be any particular upper limit, with the winning models trained at 215,000 iterations. In this way, the probability of ending up in a local minimum is significantly lowered.

[10] and sample size. This is however the dream & curse of every statistician while dealing with every problem: implicit at least

[11] Shame! (To the author's laptop of course)

7 Conclusion

As we have seen, automatically classifying images is a complex and inter-disciplinary topic, spanning from computer science to statistics. Indeed, the statistical models used to find and recognize those patterns are only a small fraction of what can (and should) be done to properly model image data. This aspect was clearly described in the pre-processing section in which the most important steps to prepare images were explained, when even the collection method influences the process. Moreover, even if the images are properly transformed, they cannot be used in their normal representation form, but some features must be extracted; the only exception would be if the statistician plans to use a convolutional neural net: however, we have seen how computationally expensive this is, even with tiny 25x25 images.

Building a neural network is also quite a complex matter: even in the extraction of features the user has multiple potential choices to make, from reducing the dimensionality of the images to extracting proper feature vectors enclosing fundamental information on the objects. Subsequently, the construction of a proper classifier poses yet another challenge. Firs of all, many different models could be used; neural networks however represent a very good alternative thanks to their ability of approximating functions of arbitrary complexity, provided enough layers and units. This type of model though present to the statistician several choices with respect to architecture: this is why for this application, six different models where fitted: it is in fact impossible to determine the best combination of hyper-parameters *a priori*: no automatic algorithm for their selection yet exists. Of course, even after several attempts, the best combination is highly unlikely to be achieved.

Nevertheless, to significantly improve performance the user must increase the complexity of the model in terms of number of input features and number of layershidden units. Indeed, the plankton specimens present a highly complex geometry where some classes are separable from others by very fine details. For this reason, image classification by features has a very limited application, probably suitable only with simple shapes. Convnets are the model of choice for whole inputs images, given that with their parameter sharing they reduce the computational time required for fitting. Moreover, also a high number of optimization epochs are required and significantly contribute to an improved performance of the model.

A Appendices

A.1 Feature Performance

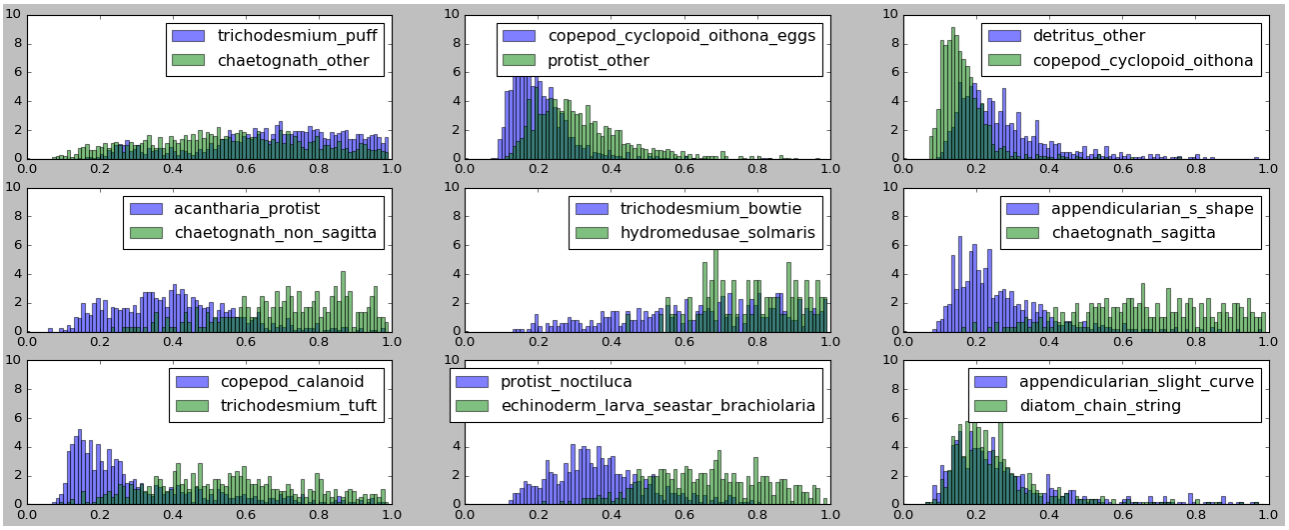


Figure 20: Distribution of *area* values among 18 classes. Note that the following values are normalized at the mean

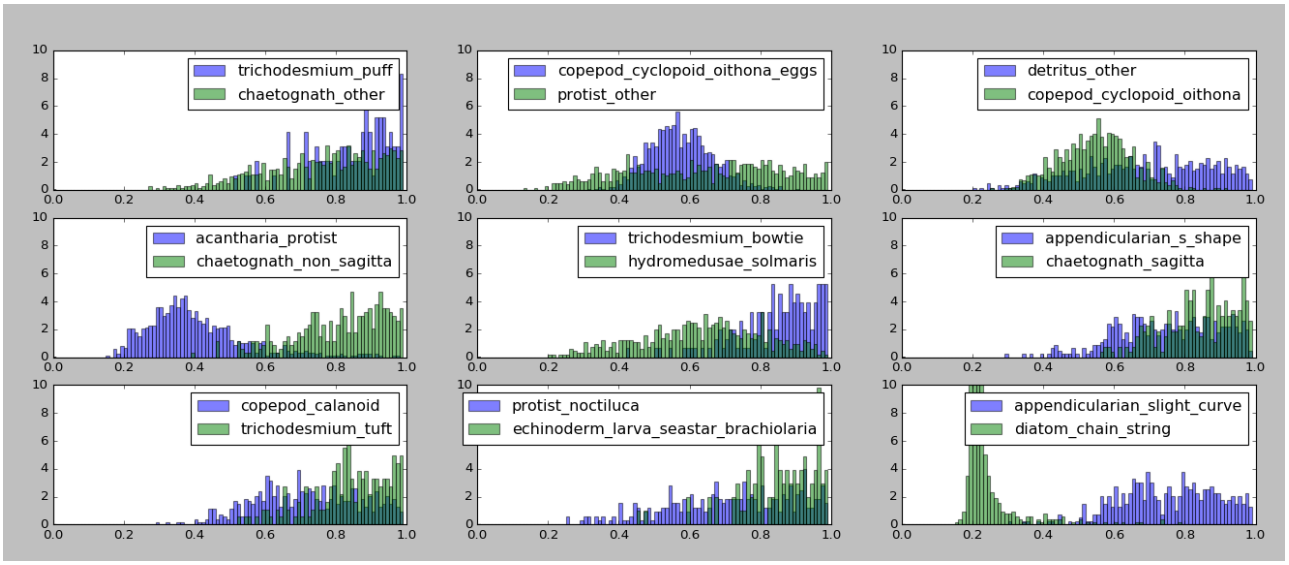


Figure 21: Distribution of the first *mean class distance* values among 18 classes. Note that the following values are normalized at the mean

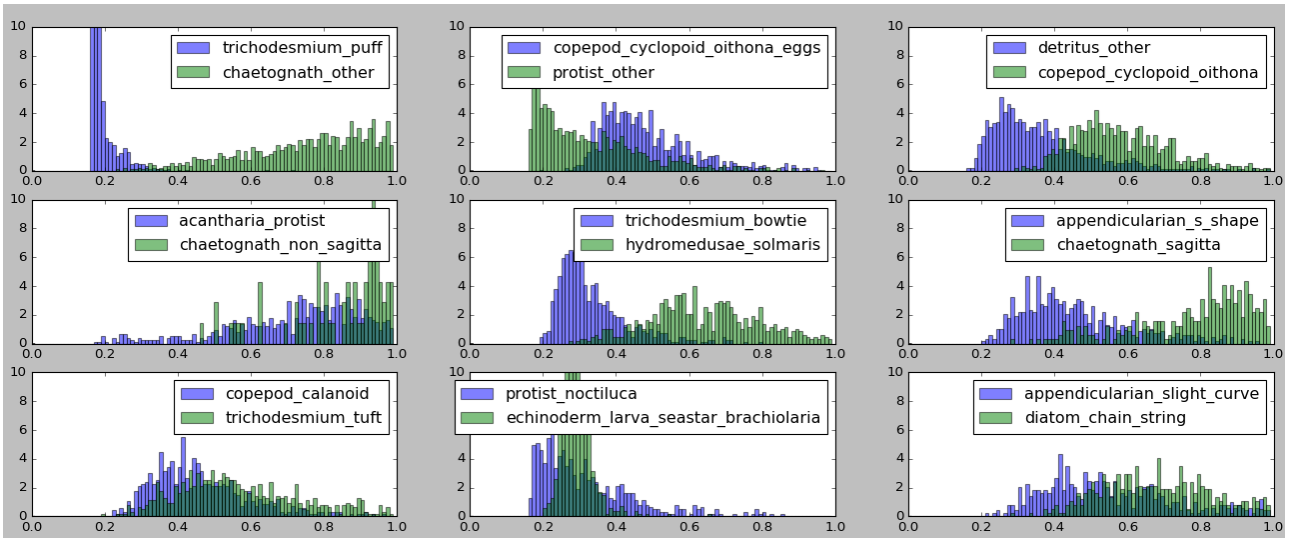


Figure 22: Distribution of the first *hu* moment values among 18 classes

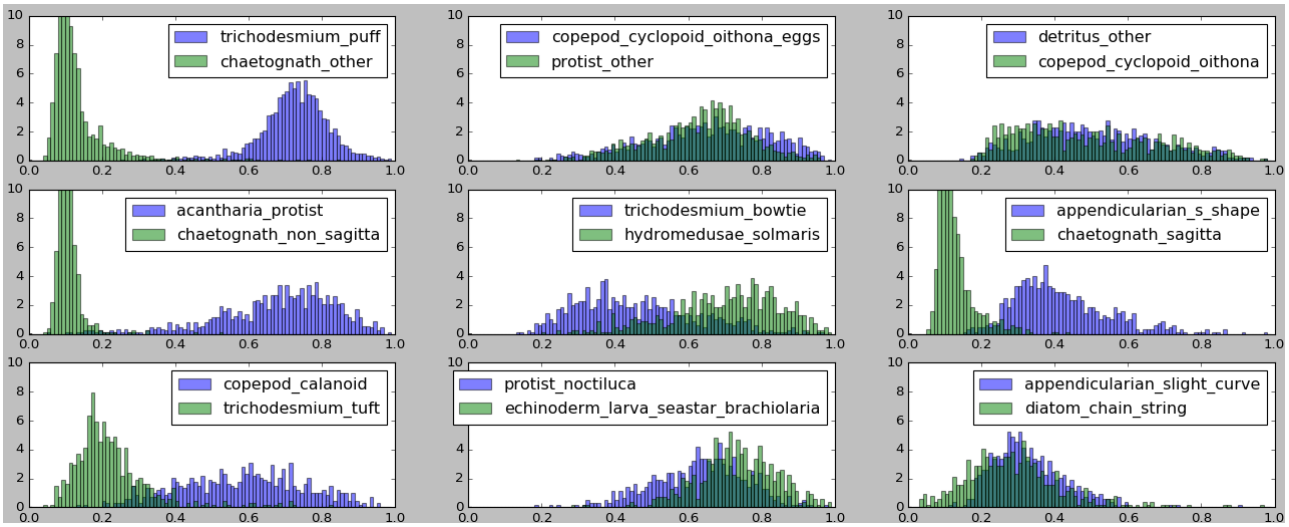


Figure 23: Distribution of *axis ratio* values among 18 classes

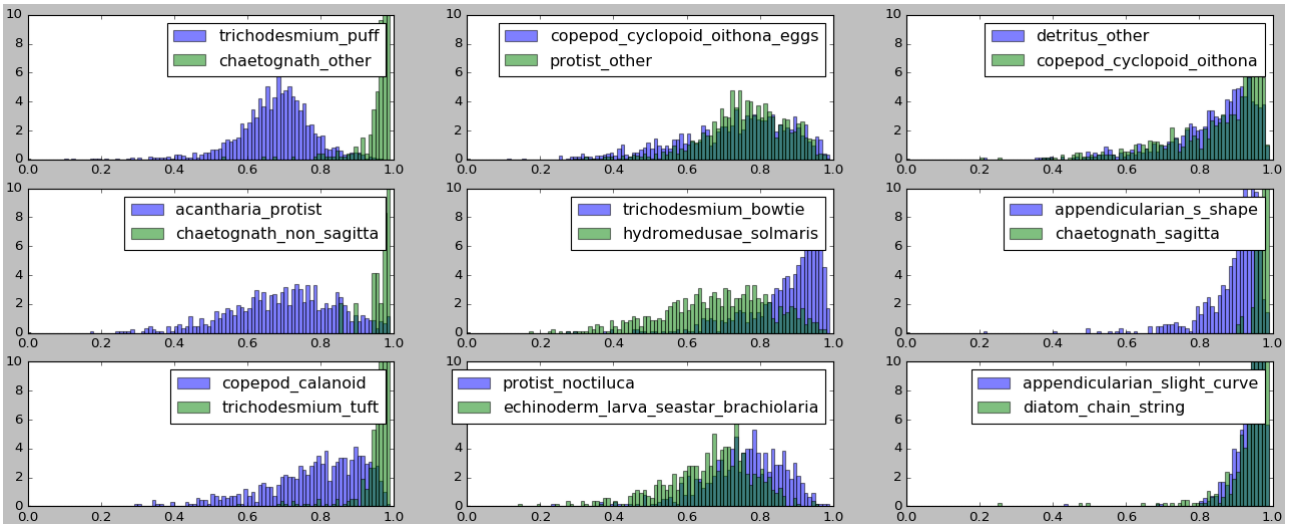


Figure 24: Distribution of the *eccentricity* values among 18 classes

A.2 Source code

A.2.1 Feed-forward NN

```
print("Importing libraries")
import time
from skimage.io import imread
from skimage.transform import resize
from sknn.mlp import Classifier, Convolution, Layer
import glob
import os
import pickle
import numpy as np
from sklearn.cross_validation import StratifiedKFold as KFold
from sklearn.metrics import classification_report, log_loss
import warnings
from skimage import morphology
from skimage import measure
warnings.filterwarnings("ignore")
print("Modules Imported")

start_time = time.time()
# ------------------------------------------------------------------------------------------------------------------------ #

print("Loading Images and functions")

dir_names = list(set(glob.glob(os.path.join("competition_data","train", "*"))\
  ).difference(set(glob.glob(os.path.join("competition_data","train","*.*")))))

dir_names = sorted(dir_names)

# Calculate the number of images in the folder
numberofImages = 0
for folder in dir_names:
    for fileNames in os.walk(folder):
        add = len(fileNames[-1])
        numberofImages += add

def getImageMeans(directory_names, pixels):

    """"# Purpose: Calculate category-wise means of images and rescale them.
    # Inputs: directory_names - a list with the location of each category folder.
    #         pixels - number of pixels to which resizing the images.
    # Output: an array with the category-wise means as row vectors."""

    # Defining initial values
    numberofClasses = np.shape(directory_names)[0]
    means_cat = np.zeros((numberofClasses, pixels**2),dtype=float) #Creates an empty array to store the means row-wise
    label = 0  # Encoding the Plankton category
    images_names = []  # Creates a vector of pictures names
    for folder in directory_names:
        for fileNameDir in os.walk(folder):
            n = 0
            arr = np.zeros((pixels, pixels), np.float)
            for fileName in fileNameDir[2]:
                if fileName[-4:] != ".jpg":
                    continue

                nameFileImage = "{0}{1}{2}".format(fileNameDir[0], os.sep, fileName)
                im = imread(nameFileImage, as_grey=True)
                im = resize(im, (pixels, pixels))  # Resizing is done
                im = np.array(im, dtype=np.float)
                images_names.append(fileName)  # Fills a vector of pictures names
```

```
            arr = arr + im
            n += 1  # Number of pictures in a folder

        mean = arr / n
        means_cat[label, 0:pixels ** 2] = np.reshape(mean,
                                                     (1, pixels ** 2)) #Matrix of Class means. One for each class.
        label += 1
    return means_cat

def getLargestRegion(props, labelmap, imagethres):

    """# Purpose: Identify the largest region in an image.
    # Inputs: props - a list with properties of each region.
    #         labelmap - list of labels of each area in the picture.
    #         imagethres - the thresholded image
    # Output: a list ."""

    regionmaxprop = None
    for regionprop in props:
        # check to see if the region is at least 50% nonzero
        if sum(imagethres[labelmap == regionprop.label])*1.0/regionprop.area < 0.50:
            continue
        if regionmaxprop is None:
            regionmaxprop = regionprop
        if regionmaxprop.filled_area < regionprop.filled_area:
            regionmaxprop = regionprop
    return regionmaxprop

def getFeatures(image):

    """# Purpose: Calculate features of a region: axis ratio, hu moments, pixels in convex area and eccentricity.
    # Inputs: image - an image.
    # Output: the features. An array"""

    image = image.copy()
    # Create the thresholded image to eliminate some of the background
    imagethr = np.where(image > np.mean(image), 0., 1.0)

    # Dilate the image
    imdilated = morphology.dilation(imagethr, np.ones((4, 4)))

    # Create the label list
    label_list = measure.label(imdilated)
    label_list = imagethr * label_list
    label_list = label_list.astype(int)

    # Get regions properties (a list with a different region features: here we use the axis length)
    region_list = measure.regionprops(label_list)
    maxregion = getLargestRegion(region_list, label_list, imagethr)

    # guard against cases where the segmentation fails by providing zeros
    ratio = 0.0
    if ((not maxregion is None) and (maxregion.major_axis_length != 0.0)):
        if maxregion is None:
            ratio = 0.0
        else:
            ratio = np.array([[maxregion.minor_axis_length * 1.0 / maxregion.major_axis_length]])
            hu = np.array([maxregion.moments_hu])
          # area = np.array([[maxregion.convex_area]])
            ecc = np.array([[maxregion.eccentricity]])
    res = np.concatenate((ratio, hu, ecc), axis= 1)
    return res
```

```
# ------------------------------------------------------------------------------------------------------------ #
# Calculating the mean images for each class.

print("Calculating class means")
means = getImageMeans(dir_names, 25)

# ------------------------------------------------------------------------------------------------------------ #
# Calculates the Y vector of labels and the X matrix of features as the  differences btw images and the classes mean

print("Getting Response and Features matrix")
pix = 25
numberofClasses = np.shape(dir_names)[0]
numFeatures = numberofClasses + 9
X = np.zeros((numberofImages, numFeatures), dtype= float)
y = np.zeros((numberofImages))
feat = np.zeros((numberofImages, 9), dtype= float)
eucl_dist = np.zeros((numberofImages, numberofClasses), dtype= float)
namesClasses = list()
images = np.zeros((numberofImages, pix*pix), dtype= float)
label = 0
i = 0

for folder in dir_names:
    currentClass = folder.split(os.pathsep)[-1]     # Creates a list of classes names as strings
    namesClasses.append(currentClass)               # Idem
    for fileNameDir in os.walk(folder):
        for fileName in fileNameDir[2]:
            if fileName[-4:] != ".jpg":
                continue
            nameFileImage = "{0}{1}{2}".format(fileNameDir[0], os.sep, fileName)
            image = imread(nameFileImage, as_grey=True)
            f = getFeatures(image)              # Get the features
            image = resize(image, (pix, pix))  # Resizing is done
            image = np.array(image, dtype=np.float)
            image = np.reshape(image, (1, pix ** 2))
            images[i, 0:pix*pix] = image
            y[i] = label                        # Vector of classes labels
            feat[i, :] = f                # Vectors of region features
            c = 0
            for t in range(0, np.shape(means)[0]):
                dist = np.linalg.norm(image - means[t, :])  # Euclidean distance
                eucl_dist[i, c] = dist  # Matrix of features
                c += 1
            i += 1
    label += 1
    print("Progress: ", label, " of 121")

X = np.concatenate((eucl_dist, feat), axis= 1)    # Matrix of input variables

# ------------------------------------------------------------------------------------------------------------ #
# Constructing and fitting the Neural Network

print("Start fitting NN")
layer_1 = Layer("Tanh", units= 180)
layer_2 = Layer("Tanh", units = 150)
layer_out = Layer("Softmax")            # Next try, add an hidden layer
lay = [layer_1, layer_2, layer_out]
nn = Classifier(layers= lay, learning_rate= 0.001, n_iter= 15) # Maybe increase the number of iterations
nn.fit(X= X, y= y)
print("Saving Model")
```

```
# Saving the model
pickle.dump(nn, open("Big_f_2_layer_15Iterations.pk1", "wb"))

# ------------------------------------------------------------------------------------------------------------------------ #
# Estimating the generalisation error with CV: all classes indivudually and multiclass log-loss

print("CV for class-wise generalisation errors")
num_folds = 2
kf = KFold(y, n_folds=num_folds)
y_pred = y * 0
l_loss = np.zeros((num_folds,1), dtype= float)
p = 0

for train, test in kf:
    X_train, X_test, y_train, y_test = X[train,:], X[test,:], y[train], y[test]
    nn_cv = Classifier(layers=lay, learning_rate=0.001, n_iter=15)
    nn_cv.fit(X=X_train, y=y_train)
    y_pred[test] = nn_cv.predict(X_test)
    y_pred2 = nn_cv.predict_proba(X_test)
    l_loss[p, 0] = log_loss(y_test, y_pred2)
    p += 1
print(classification_report(y, y_pred, target_names=namesClasses))
log_loss_CV = np.average(l_loss, axis=0)

# Calculating the multiclass log-loss
print("Multiclass Log-loss by CV: ", log_loss_CV)

print("Finished program")
print("--- %s seconds ---" % (round(time.time() - start_time, 4))) # Calculates machine time for the program
```

A.2.2 Convolutional NN

```
print("Importing libraries")
import time
from skimage.io import imread
from skimage.transform import resize
from sknn.mlp import Classifier, Convolution, Layer
import glob
import os
import pickle
import numpy as np
from sklearn.cross_validation import StratifiedKFold as KFold
from sklearn.metrics import classification_report, log_loss
import warnings
from skimage import morphology
from skimage import measure
warnings.filterwarnings("ignore")
print("Modules Imported")

start_time = time.time()
# ------------------------------------------------------------------------------------------------------------------------ #

print("Loading and preparing features datasets")

dir_names = list(set(glob.glob(os.path.join("competition_data","train", "*"))\
  ).difference(set(glob.glob(os.path.join("competition_data","train","*.*")))))

dir_names = sorted(dir_names)

# Calculate the number of images in the folder
numberofImages = 0
for folder in dir_names:
    for fileNames in os.walk(folder):
        add = len(fileNames[-1])
        numberofImages += add

# ------------------------------------------------------------------------------------------------------------------------ #
# Calculates the Y vector of labels and the X matrix of features as the  differences btw images and the classes mean

pix = 25
X = np.zeros((numberofImages, pix**2), dtype=float)
y = np.zeros((numberofImages))
namesClasses = list()
label = 0
i = 0

for folder in dir_names:
    currentClass = folder.split(os.pathsep)[-1]     # Creates a list of classes names as strings
    namesClasses.append(currentClass)               # Idem
    for fileNameDir in os.walk(folder):
        for fileName in fileNameDir[2]:
            if fileName[-4:] != ".jpg":
                continue
            nameFileImage = "{0}{1}{2}".format(fileNameDir[0], os.sep, fileName)
            image = imread(nameFileImage, as_grey=True)
            image = resize(image, (pix, pix)) # Resizing is done
            image = np.array(image, dtype=np.float)
            image = np.reshape(image, (1, pix ** 2))
            X[i, 0:pix * pix] = image
            y[i] = label
            i += 1
    label += 1
    print("Progress: ", label, " of 121")
```

```
# Fitting a convoluted neural network
print("Start fitting convoluted NN")
layer_1 = Convolution("Tanh", channels= 6, kernel_shape= (3,3))
layer_out = Layer("Softmax")
lay = [layer_1, layer_out]
nn = Classifier(layers= lay, learning_rate= 0.001, n_iter= 2)

print("Start fitting NN")
nn.fit(X= X, y= y)
print("Fineshed fitting")

# Saving the NN
pickle.dump(nn, open("Convoluted.pk1", "wb"))

# ------------------------------------------------------------------------------------------------------------------------ #
# Estimating the generalisation error with CV: all classes indivudually and multiclass log-loss

print("CV for class-wise generalisation errors")
num_folds = 2
kf = KFold(y, n_folds=num_folds)
y_pred = y * 0
l_loss = np.zeros((num_folds,1), dtype= float)
p = 0

for train, test in kf:
    X_train, X_test, y_train, y_test = X[train,:], X[test,:], y[train], y[test]
    nn_cv = Classifier(layers=lay, learning_rate=0.001, n_iter=2)
    nn_cv.fit(X=X_train, y=y_train)
    y_pred[test] = nn_cv.predict(X_test)
    y_pred2 = nn_cv.predict_proba(X_test)
    l_loss[p, 0] = log_loss(y_test, y_pred2)
    p += 1
print(classification_report(y, y_pred, target_names=namesClasses))
log_loss_CV = np.average(l_loss, axis=0)

# Calculating the multiclass log-loss
print("Multiclass Log-loss by CV: ", log_loss_CV)

print("Finished program")
print("--- %s seconds ---" % (round(time.time() - start_time, 4))) # Calculates machine time for the program
```

A.3 Model performances

A.3.1 Model One

	precision	recall	f1-score	support
acantharia_protist	0.28	0.68	0.40	889
acantharia_protist_big_center	0.00	0.00	0.00	13
acantharia_protist_halo	0.00	0.00	0.00	71
amphipods	0.00	0.00	0.00	49
appendicularian_fritillaridae	0.00	0.00	0.00	16
appendicularian_s_shape	0.11	0.06	0.08	696
appendicularian_slight_curve	0.16	0.01	0.02	532
appendicularian_straight	0.00	0.00	0.00	242
artifacts	0.14	0.01	0.01	393
artifacts_edge	0.00	0.00	0.00	170
chaetognath_non_sagitta	0.25	0.01	0.02	815
chaetognath_other	0.21	0.78	0.33	1934
chaetognath_sagitta	0.00	0.00	0.00	694
chordate_type1	0.00	0.00	0.00	77
copepod_calanoid	0.27	0.16	0.21	681
copepod_calanoid_eggs	0.00	0.00	0.00	173
copepod_calanoid_eucalanus	0.00	0.00	0.00	96
copepod_calanoid_flatheads	0.00	0.00	0.00	178
copepod_calanoid_frillyAntennae	0.00	0.00	0.00	63
copepod_calanoid_large	0.12	0.01	0.01	286
copepod_calanoid_large_side_antennatucked	0.00	0.00	0.00	106
copepod_calanoid_octomoms	0.00	0.00	0.00	49
copepod_calanoid_small_longantennae	0.00	0.00	0.00	87
copepod_cyclopoid_copilia	0.00	0.00	0.00	30
copepod_cyclopoid_oithona	0.20	0.61	0.31	899
copepod_cyclopoid_oithona_eggs	0.41	0.42	0.42	1189
copepod_other	0.00	0.00	0.00	24
crustacean_other	0.00	0.00	0.00	201
ctenophore_cestid	0.00	0.00	0.00	113
ctenophore_cydippid_no_tentacles	0.00	0.00	0.00	42
ctenophore_cydippid_tentacles	0.00	0.00	0.00	53
ctenophore_lobate	0.00	0.00	0.00	38
decapods	0.00	0.00	0.00	55
detritus_blob	0.00	0.00	0.00	363
detritus_filamentous	0.00	0.00	0.00	394
detritus_other	0.15	0.14	0.14	914
diatom_chain_string	0.40	0.85	0.54	519
diatom_chain_tube	0.00	0.00	0.00	500
echinoderm_larva_pluteus_brittlestar	0.00	0.00	0.00	36
echinoderm_larva_pluteus_early	0.00	0.00	0.00	92
echinoderm_larva_pluteus_typeC	0.00	0.00	0.00	80
echinoderm_larva_pluteus_urchin	0.00	0.00	0.00	88
echinoderm_larva_seastar_bipinnaria	0.21	0.57	0.31	385
echinoderm_larva_seastar_brachiolaria	0.25	0.14	0.18	536
echinoderm_seacucumber_auricularia_larva	0.00	0.00	0.00	96
echinopluteus	0.00	0.00	0.00	27
ephyra	0.00	0.00	0.00	14
euphausiids	0.00	0.00	0.00	136
euphausiids_young	0.00	0.00	0.00	38
fecal_pellet	0.00	0.00	0.00	511
fish_larvae_deep_body	0.00	0.00	0.00	10
fish_larvae_leptocephali	0.00	0.00	0.00	31
fish_larvae_medium_body	0.00	0.00	0.00	85
fish_larvae_myctophids	0.00	0.00	0.00	114
fish_larvae_thin_body	0.00	0.00	0.00	64
fish_larvae_very_thin_body	0.00	0.00	0.00	16
heteropod	0.00	0.00	0.00	10
hydromedusae_aglaura	0.00	0.00	0.00	127

hydromedusae_bell_and_tentacles	0.00	0.00	0.00	75
hydromedusae_h15	0.00	0.00	0.00	35
hydromedusae_haliscera	0.20	0.10	0.13	229
hydromedusae_haliscera_small_sideview	0.00	0.00	0.00	9
hydromedusae_liriope	0.00	0.00	0.00	19
hydromedusae_narco_dark	0.00	0.00	0.00	23
hydromedusae_narco_young	0.00	0.00	0.00	336
hydromedusae_narcomedusae	0.00	0.00	0.00	132
hydromedusae_other	0.00	0.00	0.00	12
hydromedusae_partial_dark	0.00	0.00	0.00	190
hydromedusae_shapeA	0.18	0.53	0.27	412
hydromedusae_shapeA_sideview_small	0.00	0.00	0.00	274
hydromedusae_shapeB	0.00	0.00	0.00	150
hydromedusae_sideview_big	0.00	0.00	0.00	76
hydromedusae_solmaris	0.13	0.22	0.16	703
hydromedusae_solmundella	0.00	0.00	0.00	123
hydromedusae_typeD	0.00	0.00	0.00	43
hydromedusae_typeD_bell_and_tentacles	0.00	0.00	0.00	56
hydromedusae_typeE	0.00	0.00	0.00	14
hydromedusae_typeF	0.00	0.00	0.00	61
invertebrate_larvae_other_A	0.00	0.00	0.00	14
invertebrate_larvae_other_B	0.00	0.00	0.00	24
jellies_tentacles	0.00	0.00	0.00	141
polychaete	0.00	0.00	0.00	131
protist_dark_center	0.00	0.00	0.00	108
protist_fuzzy_olive	1.00	0.03	0.06	372
protist_noctiluca	0.00	0.00	0.00	625
protist_other	0.19	0.63	0.29	1172
protist_star	0.00	0.00	0.00	113
pteropod_butterfly	0.00	0.00	0.00	108
pteropod_theco_dev_seq	0.00	0.00	0.00	13
pteropod_triangle	0.00	0.00	0.00	65
radiolarian_chain	0.00	0.00	0.00	287
radiolarian_colony	0.00	0.00	0.00	158
shrimp-like_other	0.00	0.00	0.00	52
shrimp_caridean	0.00	0.00	0.00	49
shrimp_sergestidae	0.00	0.00	0.00	153
shrimp_zoea	0.00	0.00	0.00	174
siphonophore_calycophoran_abylidae	0.00	0.00	0.00	212
siphonophore_calycophoran_rocketship_adult	0.00	0.00	0.00	135
siphonophore_calycophoran_rocketship_young	0.00	0.00	0.00	483
siphonophore_calycophoran_sphaeronectes	0.00	0.00	0.00	179
siphonophore_calycophoran_sphaeronectes_stem	0.00	0.00	0.00	57
siphonophore_calycophoran_sphaeronectes_young	0.00	0.00	0.00	247
siphonophore_other_parts	0.00	0.00	0.00	29
siphonophore_partial	0.00	0.00	0.00	30
siphonophore_physonect	0.00	0.00	0.00	128
siphonophore_physonect_young	0.00	0.00	0.00	21
stomatopod	0.00	0.00	0.00	24
tornaria_acorn_worm_larvae	0.00	0.00	0.00	38
trichodesmium_bowtie	0.17	0.23	0.20	708
trichodesmium_multiple	0.00	0.00	0.00	54
trichodesmium_puff	0.63	0.85	0.73	1979
trichodesmium_tuft	0.07	0.16	0.10	678
trochophore_larvae	0.00	0.00	0.00	29
tunicate_doliolid	0.00	0.00	0.00	439
tunicate_doliolid_nurse	0.00	0.00	0.00	417
tunicate_partial	0.41	0.80	0.54	352
tunicate_salp	0.28	0.31	0.30	236
tunicate_salp_chains	0.00	0.00	0.00	73
unknown_blobs_and_smudges	0.00	0.00	0.00	317
unknown_sticks	0.00	0.00	0.00	175
unknown_unclassified	0.00	0.00	0.00	425
avg / total	0.16	0.25	0.17	30336

A.3.2 Model Two

	precision	recall	f1−score	support
acantharia_protist	0.25	0.57	0.35	889
acantharia_protist_big_center	0.00	0.00	0.00	13
acantharia_protist_halo	0.00	0.00	0.00	71
amphipods	0.00	0.00	0.00	49
appendicularian_fritillaridae	0.00	0.00	0.00	16
appendicularian_s_shape	0.14	0.03	0.05	696
appendicularian_slight_curve	0.00	0.00	0.00	532
appendicularian_straight	0.00	0.00	0.00	242
artifacts	0.00	0.00	0.00	393
artifacts_edge	0.00	0.00	0.00	170
chaetognath_non_sagitta	0.06	0.01	0.01	815
chaetognath_other	0.17	0.76	0.28	1934
chaetognath_sagitta	0.00	0.00	0.00	694
chordate_type1	0.00	0.00	0.00	77
copepod_calanoid	0.30	0.07	0.11	681
copepod_calanoid_eggs	0.00	0.00	0.00	173
copepod_calanoid_eucalanus	0.00	0.00	0.00	96
copepod_calanoid_flatheads	0.00	0.00	0.00	178
copepod_calanoid_frillyAntennae	0.00	0.00	0.00	63
copepod_calanoid_large	0.00	0.00	0.00	286
copepod_calanoid_large_side_antennatucked	0.00	0.00	0.00	106
copepod_calanoid_octomoms	0.00	0.00	0.00	49
copepod_calanoid_small_longantennae	0.00	0.00	0.00	87
copepod_cyclopoid_copilia	0.00	0.00	0.00	30
copepod_cyclopoid_oithona	0.15	0.16	0.16	899
copepod_cyclopoid_oithona_eggs	0.25	0.84	0.39	1189
copepod_other	0.00	0.00	0.00	24
crustacean_other	0.00	0.00	0.00	201
ctenophore_cestid	0.00	0.00	0.00	113
ctenophore_cydippid_no_tentacles	0.00	0.00	0.00	42
ctenophore_cydippid_tentacles	0.00	0.00	0.00	53
ctenophore_lobate	0.00	0.00	0.00	38
decapods	0.00	0.00	0.00	55
detritus_blob	0.00	0.00	0.00	363
detritus_filamentous	0.00	0.00	0.00	394
detritus_other	0.13	0.09	0.10	914
diatom_chain_string	0.59	0.67	0.63	519
diatom_chain_tube	0.00	0.00	0.00	500
echinoderm_larva_pluteus_brittlestar	0.00	0.00	0.00	36
echinoderm_larva_pluteus_early	0.00	0.00	0.00	92
echinoderm_larva_pluteus_typeC	0.00	0.00	0.00	80
echinoderm_larva_pluteus_urchin	0.00	0.00	0.00	88
echinoderm_larva_seastar_bipinnaria	0.25	0.05	0.08	385
echinoderm_larva_seastar_brachiolaria	0.14	0.24	0.17	536
echinoderm_seacucumber_auricularia_larva	0.00	0.00	0.00	96
echinopluteus	0.00	0.00	0.00	27
ephyra	0.00	0.00	0.00	14
euphausiids	0.00	0.00	0.00	136
euphausiids_young	0.00	0.00	0.00	38
fecal_pellet	0.00	0.00	0.00	511
fish_larvae_deep_body	0.00	0.00	0.00	10
fish_larvae_leptocephali	0.00	0.00	0.00	31
fish_larvae_medium_body	0.00	0.00	0.00	85
fish_larvae_myctophids	0.00	0.00	0.00	114
fish_larvae_thin_body	0.00	0.00	0.00	64
fish_larvae_very_thin_body	0.00	0.00	0.00	16
heteropod	0.00	0.00	0.00	10
hydromedusae_aglaura	0.00	0.00	0.00	127
hydromedusae_bell_and_tentacles	0.00	0.00	0.00	75
hydromedusae_h15	0.00	0.00	0.00	35
hydromedusae_haliscera	0.00	0.00	0.00	229

hydromedusae_haliscera_small_sideview	0.00	0.00	0.00	9
hydromedusae_liriope	0.00	0.00	0.00	19
hydromedusae_narco_dark	0.00	0.00	0.00	23
hydromedusae_narco_young	0.00	0.00	0.00	336
hydromedusae_narcomedusae	0.00	0.00	0.00	132
hydromedusae_other	0.00	0.00	0.00	12
hydromedusae_partial_dark	0.00	0.00	0.00	190
hydromedusae_shapeA	0.18	0.22	0.20	412
hydromedusae_shapeA_sideview_small	0.00	0.00	0.00	274
hydromedusae_shapeB	0.00	0.00	0.00	150
hydromedusae_sideview_big	0.00	0.00	0.00	76
hydromedusae_solmaris	0.10	0.34	0.15	703
hydromedusae_solmundella	0.00	0.00	0.00	123
hydromedusae_typeD	0.00	0.00	0.00	43
hydromedusae_typeD_bell_and_tentacles	0.00	0.00	0.00	56
hydromedusae_typeE	0.00	0.00	0.00	14
hydromedusae_typeF	0.00	0.00	0.00	61
invertebrate_larvae_other_A	0.00	0.00	0.00	14
invertebrate_larvae_other_B	0.00	0.00	0.00	24
jellies_tentacles	0.00	0.00	0.00	141
polychaete	0.00	0.00	0.00	131
protist_dark_center	0.00	0.00	0.00	108
protist_fuzzy_olive	0.00	0.00	0.00	372
protist_noctiluca	0.17	0.02	0.03	625
protist_other	0.21	0.33	0.26	1172
protist_star	0.00	0.00	0.00	113
pteropod_butterfly	0.00	0.00	0.00	108
pteropod_theco_dev_seq	0.00	0.00	0.00	13
pteropod_triangle	0.00	0.00	0.00	65
radiolarian_chain	0.00	0.00	0.00	287
radiolarian_colony	0.00	0.00	0.00	158
shrimp-like_other	0.00	0.00	0.00	52
shrimp_caridean	0.00	0.00	0.00	49
shrimp_sergestidae	0.00	0.00	0.00	153
shrimp_zoea	0.00	0.00	0.00	174
siphonophore_calycophoran_abylidae	0.00	0.00	0.00	212
siphonophore_calycophoran_rocketship_adult	0.00	0.00	0.00	135
siphonophore_calycophoran_rocketship_young	0.00	0.00	0.00	483
siphonophore_calycophoran_sphaeronectes	0.00	0.00	0.00	179
siphonophore_calycophoran_sphaeronectes_stem	0.00	0.00	0.00	57
siphonophore_calycophoran_sphaeronectes_young	0.00	0.00	0.00	247
siphonophore_other_parts	0.00	0.00	0.00	29
siphonophore_partial	0.00	0.00	0.00	30
siphonophore_physonect	0.00	0.00	0.00	128
siphonophore_physonect_young	0.00	0.00	0.00	21
stomatopod	0.00	0.00	0.00	24
tornaria_acorn_worm_larvae	0.00	0.00	0.00	38
trichodesmium_bowtie	0.12	0.12	0.12	708
trichodesmium_multiple	0.00	0.00	0.00	54
trichodesmium_puff	0.40	0.92	0.56	1979
trichodesmium_tuft	0.06	0.03	0.04	678
trochophore_larvae	0.00	0.00	0.00	29
tunicate_doliolid	0.00	0.00	0.00	439
tunicate_doliolid_nurse	0.00	0.00	0.00	417
tunicate_partial	0.18	0.94	0.30	352
tunicate_salp	0.00	0.00	0.00	236
tunicate_salp_chains	0.00	0.00	0.00	73
unknown_blobs_and_smudges	0.00	0.00	0.00	317
unknown_sticks	0.00	0.00	0.00	175
unknown_unclassified	0.00	0.00	0.00	425
avg / total	0.11	0.22	0.13	30336

A.3.3 Model Three

	precision	recall	f1-score	support
acantharia_protist	0.42	0.69	0.52	889
acantharia_protist_big_center	0.00	0.00	0.00	13
acantharia_protist_halo	0.00	0.00	0.00	71
amphipods	0.00	0.00	0.00	49
appendicularian_fritillaridae	0.00	0.00	0.00	16
appendicularian_s_shape	0.16	0.30	0.21	696
appendicularian_slight_curve	0.11	0.16	0.13	532
appendicularian_straight	0.00	0.00	0.00	242
artifacts	0.26	0.46	0.33	393
artifacts_edge	0.41	0.14	0.21	170
chaetognath_non_sagitta	0.36	0.26	0.30	815
chaetognath_other	0.33	0.73	0.45	1934
chaetognath_sagitta	0.00	0.00	0.00	694
chordate_type1	0.18	0.48	0.27	77
copepod_calanoid	0.22	0.65	0.32	681
copepod_calanoid_eggs	0.00	0.00	0.00	173
copepod_calanoid_eucalanus	0.00	0.00	0.00	96
copepod_calanoid_flatheads	0.00	0.00	0.00	178
copepod_calanoid_frillyAntennae	0.00	0.00	0.00	63
copepod_calanoid_large	0.36	0.13	0.19	286
copepod_calanoid_large_side_antennatucked	0.00	0.00	0.00	106
copepod_calanoid_octomoms	0.00	0.00	0.00	49
copepod_calanoid_small_longantennae	0.00	0.00	0.00	87
copepod_cyclopoid_copilia	0.00	0.00	0.00	30
copepod_cyclopoid_oithona	0.33	0.36	0.34	899
copepod_cyclopoid_oithona_eggs	0.46	0.64	0.54	1189
copepod_other	0.00	0.00	0.00	24
crustacean_other	0.11	0.04	0.06	201
ctenophore_cestid	0.12	0.02	0.03	113
ctenophore_cydippid_no_tentacles	0.00	0.00	0.00	42
ctenophore_cydippid_tentacles	0.00	0.00	0.00	53
ctenophore_lobate	0.00	0.00	0.00	38
decapods	0.00	0.00	0.00	55
detritus_blob	0.09	0.09	0.09	363
detritus_filamentous	0.00	0.00	0.00	394
detritus_other	0.18	0.15	0.17	914
diatom_chain_string	0.69	0.82	0.75	519
diatom_chain_tube	0.42	0.11	0.18	500
echinoderm_larva_pluteus_brittlestar	0.00	0.00	0.00	36
echinoderm_larva_pluteus_early	0.00	0.00	0.00	92
echinoderm_larva_pluteus_typeC	0.00	0.00	0.00	80
echinoderm_larva_pluteus_urchin	0.00	0.00	0.00	88
echinoderm_larva_seastar_bipinnaria	0.49	0.38	0.43	385
echinoderm_larva_seastar_brachiolaria	0.45	0.62	0.52	536
echinoderm_seacucumber_auricularia_larva	0.00	0.00	0.00	96
echinopluteus	0.00	0.00	0.00	27
ephyra	0.00	0.00	0.00	14
euphausiids	0.00	0.00	0.00	136
euphausiids_young	0.00	0.00	0.00	38
fecal_pellet	0.00	0.00	0.00	511
fish_larvae_deep_body	0.00	0.00	0.00	10
fish_larvae_leptocephali	0.00	0.00	0.00	31
fish_larvae_medium_body	0.10	0.32	0.16	85
fish_larvae_myctophids	0.14	0.03	0.04	114
fish_larvae_thin_body	0.00	0.00	0.00	64
fish_larvae_very_thin_body	0.00	0.00	0.00	16
heteropod	0.00	0.00	0.00	10
hydromedusae_aglaura	0.00	0.00	0.00	127
hydromedusae_bell_and_tentacles	0.00	0.00	0.00	75
hydromedusae_h15	0.00	0.00	0.00	35
hydromedusae_haliscera	0.25	0.21	0.23	229
hydromedusae_haliscera_small_sideview	0.00	0.00	0.00	9

hydromedusae_liriope	0.00	0.00	0.00	19
hydromedusae_narco_dark	0.00	0.00	0.00	23
hydromedusae_narco_young	0.14	0.00	0.01	336
hydromedusae_narcomedusae	0.00	0.00	0.00	132
hydromedusae_other	0.00	0.00	0.00	12
hydromedusae_partial_dark	0.30	0.34	0.32	190
hydromedusae_shapeA	0.19	0.60	0.29	412
hydromedusae_shapeA_sideview_small	0.00	0.00	0.00	274
hydromedusae_shapeB	0.00	0.00	0.00	150
hydromedusae_sideview_big	0.00	0.00	0.00	76
hydromedusae_solmaris	0.17	0.41	0.24	703
hydromedusae_solmundella	0.00	0.00	0.00	123
hydromedusae_typeD	0.00	0.00	0.00	43
hydromedusae_typeD_bell_and_tentacles	0.00	0.00	0.00	56
hydromedusae_typeE	0.00	0.00	0.00	14
hydromedusae_typeF	0.05	0.02	0.02	61
invertebrate_larvae_other_A	0.00	0.00	0.00	14
invertebrate_larvae_other_B	0.00	0.00	0.00	24
jellies_tentacles	0.00	0.00	0.00	141
polychaete	0.00	0.00	0.00	131
protist_dark_center	0.00	0.00	0.00	108
protist_fuzzy_olive	0.78	0.50	0.61	372
protist_noctiluca	0.23	0.19	0.20	625
protist_other	0.25	0.65	0.37	1172
protist_star	0.41	0.37	0.39	113
pteropod_butterfly	0.00	0.00	0.00	108
pteropod_theco_dev_seq	0.00	0.00	0.00	13
pteropod_triangle	0.00	0.00	0.00	65
radiolarian_chain	0.33	0.00	0.01	287
radiolarian_colony	0.14	0.05	0.07	158
shrimp-like_other	0.00	0.00	0.00	52
shrimp_caridean	0.00	0.00	0.00	49
shrimp_sergestidae	0.00	0.00	0.00	153
shrimp_zoea	0.00	0.00	0.00	174
siphonophore_calycophoran_abylidae	0.09	0.01	0.02	212
siphonophore_calycophoran_rocketship_adult	0.00	0.00	0.00	135
siphonophore_calycophoran_rocketship_young	0.50	0.00	0.00	483
siphonophore_calycophoran_sphaeronectes	0.17	0.05	0.08	179
siphonophore_calycophoran_sphaeronectes_stem	0.00	0.00	0.00	57
siphonophore_calycophoran_sphaeronectes_young	0.00	0.00	0.00	247
siphonophore_other_parts	0.00	0.00	0.00	29
siphonophore_partial	0.00	0.00	0.00	30
siphonophore_physonect	0.00	0.00	0.00	128
siphonophore_physonect_young	0.00	0.00	0.00	21
stomatopod	0.00	0.00	0.00	24
tornaria_acorn_worm_larvae	0.00	0.00	0.00	38
trichodesmium_bowtie	0.29	0.26	0.27	708
trichodesmium_multiple	0.00	0.00	0.00	54
trichodesmium_puff	0.80	0.85	0.82	1979
trichodesmium_tuft	0.16	0.37	0.23	678
trochophore_larvae	0.00	0.00	0.00	29
tunicate_doliolid	0.40	0.01	0.02	439
tunicate_doliolid_nurse	0.06	0.01	0.02	417
tunicate_partial	0.64	0.71	0.67	352
tunicate_salp	0.29	0.58	0.39	236
tunicate_salp_chains	0.00	0.00	0.00	73
unknown_blobs_and_smudges	0.06	0.03	0.05	317
unknown_sticks	0.00	0.00	0.00	175
unknown_unclassified	0.00	0.00	0.00	425
avg / total	0.26	0.32	0.26	30336

A.3.4 Model Four

	precision	recall	f1-score	support
acantharia_protist	0.00	0.00	0.00	889
acantharia_protist_big_center	0.00	0.00	0.00	13
acantharia_protist_halo	0.00	0.00	0.00	71
amphipods	0.00	0.00	0.00	49
appendicularian_fritillaridae	0.00	0.00	0.00	16
appendicularian_s_shape	0.00	0.00	0.00	696
appendicularian_slight_curve	0.00	0.00	0.00	532
appendicularian_straight	0.00	0.00	0.00	242
artifacts	0.00	0.00	0.00	393
artifacts_edge	0.00	0.00	0.00	170
chaetognath_non_sagitta	0.00	0.00	0.00	815
chaetognath_other	0.33	0.00	0.00	1934
chaetognath_sagitta	0.00	0.00	0.00	694
chordate_type1	0.00	0.00	0.00	77
copepod_calanoid	0.00	0.00	0.00	681
copepod_calanoid_eggs	0.00	0.00	0.00	173
copepod_calanoid_eucalanus	0.00	0.00	0.00	96
copepod_calanoid_flatheads	0.00	0.00	0.00	178
copepod_calanoid_frillyAntennae	0.00	0.00	0.00	63
copepod_calanoid_large	0.00	0.00	0.00	286
copepod_calanoid_large_side_antennatucked	0.00	0.00	0.00	106
copepod_calanoid_octomoms	0.00	0.00	0.00	49
copepod_calanoid_small_longantennae	0.00	0.00	0.00	87
copepod_cyclopoid_copilia	0.00	0.00	0.00	30
copepod_cyclopoid_oithona	0.00	0.00	0.00	899
copepod_cyclopoid_oithona_eggs	0.00	0.00	0.00	1189
copepod_other	0.00	0.00	0.00	24
crustacean_other	0.00	0.00	0.00	201
ctenophore_cestid	0.00	0.00	0.00	113
ctenophore_cydippid_no_tentacles	0.00	0.00	0.00	42
ctenophore_cydippid_tentacles	0.00	0.00	0.00	53
ctenophore_lobate	0.00	0.00	0.00	38
decapods	0.00	0.00	0.00	55
detritus_blob	0.00	0.00	0.00	363
detritus_filamentous	0.00	0.00	0.00	394
detritus_other	0.00	0.00	0.00	914
diatom_chain_string	0.00	0.00	0.00	519
diatom_chain_tube	0.00	0.00	0.00	500
echinoderm_larva_pluteus_brittlestar	0.00	0.00	0.00	36
echinoderm_larva_pluteus_early	0.00	0.00	0.00	92
echinoderm_larva_pluteus_typeC	0.00	0.00	0.00	80
echinoderm_larva_pluteus_urchin	0.00	0.00	0.00	88
echinoderm_larva_seastar_bipinnaria	0.00	0.00	0.00	385
echinoderm_larva_seastar_brachiolaria	0.00	0.00	0.00	536
echinoderm_seacucumber_auricularia_larva	0.00	0.00	0.00	96
echinopluteus	0.00	0.00	0.00	27
ephyra	0.00	0.00	0.00	14
euphausiids	0.00	0.00	0.00	136
euphausiids_young	0.00	0.00	0.00	38
fecal_pellet	0.00	0.00	0.00	511
fish_larvae_deep_body	0.00	0.00	0.00	10
fish_larvae_leptocephali	0.00	0.00	0.00	31
fish_larvae_medium_body	0.00	0.00	0.00	85
fish_larvae_myctophids	0.00	0.00	0.00	114
fish_larvae_thin_body	0.00	0.00	0.00	64
fish_larvae_very_thin_body	0.00	0.00	0.00	16
heteropod	0.00	0.00	0.00	10
hydromedusae_aglaura	0.00	0.00	0.00	127
hydromedusae_bell_and_tentacles	0.00	0.00	0.00	75
hydromedusae_h15	0.00	0.00	0.00	35
hydromedusae_haliscera	0.00	0.00	0.00	229
hydromedusae_haliscera_small_sideview	0.00	0.00	0.00	9

hydromedusae_liriope	0.00	0.00	0.00	19
hydromedusae_narco_dark	0.00	0.00	0.00	23
hydromedusae_narco_young	0.00	0.00	0.00	336
hydromedusae_narcomedusae	0.00	0.00	0.00	132
hydromedusae_other	0.00	0.00	0.00	12
hydromedusae_partial_dark	0.00	0.00	0.00	190
hydromedusae_shapeA	0.00	0.00	0.00	412
hydromedusae_shapeA_sideview_small	0.00	0.00	0.00	274
hydromedusae_shapeB	0.00	0.00	0.00	150
hydromedusae_sideview_big	0.00	0.00	0.00	76
hydromedusae_solmaris	0.00	0.00	0.00	703
hydromedusae_solmundella	0.00	0.00	0.00	123
hydromedusae_typeD	0.00	0.00	0.00	43
hydromedusae_typeD_bell_and_tentacles	0.00	0.00	0.00	56
hydromedusae_typeE	0.00	0.00	0.00	14
hydromedusae_typeF	0.00	0.00	0.00	61
invertebrate_larvae_other_A	0.00	0.00	0.00	14
invertebrate_larvae_other_B	0.00	0.00	0.00	24
jellies_tentacles	0.00	0.00	0.00	141
polychaete	0.00	0.00	0.00	131
protist_dark_center	0.00	0.00	0.00	108
protist_fuzzy_olive	0.00	0.00	0.00	372
protist_noctiluca	0.00	0.00	0.00	625
protist_other	0.04	1.00	0.07	1172
protist_star	0.00	0.00	0.00	113
pteropod_butterfly	0.00	0.00	0.00	108
pteropod_theco_dev_seq	0.00	0.00	0.00	13
pteropod_triangle	0.00	0.00	0.00	65
radiolarian_chain	0.00	0.00	0.00	287
radiolarian_colony	0.00	0.00	0.00	158
shrimp-like_other	0.00	0.00	0.00	52
shrimp_caridean	0.00	0.00	0.00	49
shrimp_sergestidae	0.00	0.00	0.00	153
shrimp_zoea	0.00	0.00	0.00	174
siphonophore_calycophoran_abylidae	0.00	0.00	0.00	212
siphonophore_calycophoran_rocketship_adult	0.00	0.00	0.00	135
siphonophore_calycophoran_rocketship_young	0.00	0.00	0.00	483
siphonophore_calycophoran_sphaeronectes	0.00	0.00	0.00	179
siphonophore_calycophoran_sphaeronectes_stem	0.00	0.00	0.00	57
siphonophore_calycophoran_sphaeronectes_young	0.00	0.00	0.00	247
siphonophore_other_parts	0.00	0.00	0.00	29
siphonophore_partial	0.00	0.00	0.00	30
siphonophore_physonect	0.00	0.00	0.00	128
siphonophore_physonect_young	0.00	0.00	0.00	21
stomatopod	0.00	0.00	0.00	24
tornaria_acorn_worm_larvae	0.00	0.00	0.00	38
trichodesmium_bowtie	0.00	0.00	0.00	708
trichodesmium_multiple	0.00	0.00	0.00	54
trichodesmium_puff	0.00	0.00	0.00	1979
trichodesmium_tuft	0.00	0.00	0.00	678
trochophore_larvae	0.00	0.00	0.00	29
tunicate_doliolid	0.00	0.00	0.00	439
tunicate_doliolid_nurse	0.00	0.00	0.00	417
tunicate_partial	0.00	0.00	0.00	352
tunicate_salp	0.00	0.00	0.00	236
tunicate_salp_chains	0.00	0.00	0.00	73
unknown_blobs_and_smudges	0.00	0.00	0.00	317
unknown_sticks	0.00	0.00	0.00	175
unknown_unclassified	0.00	0.00	0.00	425
avg / total	0.02	0.04	0.00	30336

A.3.5 Model Five

	precision	recall	f1-score	support
acantharia_protist	0.60	0.56	0.58	889
acantharia_protist_big_center	0.00	0.00	0.00	13
acantharia_protist_halo	0.00	0.00	0.00	71
amphipods	0.00	0.00	0.00	49
appendicularian_fritillaridae	0.00	0.00	0.00	16
appendicularian_s_shape	0.15	0.30	0.20	696
appendicularian_slight_curve	0.17	0.14	0.15	532
appendicularian_straight	0.00	0.00	0.00	242
artifacts	0.36	0.39	0.38	393
artifacts_edge	0.48	0.25	0.33	170
chaetognath_non_sagitta	0.43	0.41	0.42	815
chaetognath_other	0.39	0.58	0.47	1934
chaetognath_sagitta	0.29	0.04	0.07	694
chordate_type1	0.15	0.75	0.24	77
copepod_calanoid	0.25	0.54	0.34	681
copepod_calanoid_eggs	0.00	0.00	0.00	173
copepod_calanoid_eucalanus	0.00	0.00	0.00	96
copepod_calanoid_flatheads	0.00	0.00	0.00	178
copepod_calanoid_frillyAntennae	0.00	0.00	0.00	63
copepod_calanoid_large	0.20	0.44	0.27	286
copepod_calanoid_large_side_antennatucked	0.00	0.00	0.00	106
copepod_calanoid_octomoms	0.00	0.00	0.00	49
copepod_calanoid_small_longantennae	0.00	0.00	0.00	87
copepod_cyclopoid_copilia	0.00	0.00	0.00	30
copepod_cyclopoid_oithona	0.27	0.57	0.37	899
copepod_cyclopoid_oithona_eggs	0.45	0.75	0.56	1189
copepod_other	0.00	0.00	0.00	24
crustacean_other	0.06	0.10	0.07	201
ctenophore_cestid	0.17	0.04	0.07	113
ctenophore_cydippid_no_tentacles	0.00	0.00	0.00	42
ctenophore_cydippid_tentacles	0.00	0.00	0.00	53
ctenophore_lobate	0.12	0.24	0.16	38
decapods	0.00	0.00	0.00	55
detritus_blob	0.11	0.07	0.08	363
detritus_filamentous	0.00	0.00	0.00	394
detritus_other	0.16	0.22	0.18	914
diatom_chain_string	0.78	0.69	0.73	519
diatom_chain_tube	0.31	0.33	0.32	500
echinoderm_larva_pluteus_brittlestar	0.00	0.00	0.00	36
echinoderm_larva_pluteus_early	0.00	0.00	0.00	92
echinoderm_larva_pluteus_typeC	0.00	0.00	0.00	80
echinoderm_larva_pluteus_urchin	0.00	0.00	0.00	88
echinoderm_larva_seastar_bipinnaria	0.36	0.68	0.47	385
echinoderm_larva_seastar_brachiolaria	0.32	0.77	0.46	536
echinoderm_seacucumber_auricularia_larva	0.00	0.00	0.00	96
echinopluteus	0.00	0.00	0.00	27
ephyra	0.00	0.00	0.00	14
euphausiids	0.20	0.01	0.01	136
euphausiids_young	0.00	0.00	0.00	38
fecal_pellet	0.00	0.00	0.00	511
fish_larvae_deep_body	0.00	0.00	0.00	10
fish_larvae_leptocephali	0.00	0.00	0.00	31
fish_larvae_medium_body	0.12	0.01	0.02	85
fish_larvae_myctophids	0.21	0.06	0.09	114
fish_larvae_thin_body	0.00	0.00	0.00	64
fish_larvae_very_thin_body	0.00	0.00	0.00	16
heteropod	0.00	0.00	0.00	10
hydromedusae_aglaura	0.00	0.00	0.00	127
hydromedusae_bell_and_tentacles	0.14	0.03	0.04	75
hydromedusae_h15	0.00	0.00	0.00	35
hydromedusae_haliscera	0.55	0.18	0.28	229
hydromedusae_haliscera_small_sideview	0.00	0.00	0.00	9
hydromedusae_liriope	0.00	0.00	0.00	19

hydromedusae_narco_dark	0.00	0.00	0.00	23
hydromedusae_narco_young	0.11	0.08	0.09	336
hydromedusae_narcomedusae	0.00	0.00	0.00	132
hydromedusae_other	0.00	0.00	0.00	12
hydromedusae_partial_dark	0.25	0.28	0.26	190
hydromedusae_shapeA	0.30	0.48	0.37	412
hydromedusae_shapeA_sideview_small	0.05	0.02	0.03	274
hydromedusae_shapeB	0.10	0.01	0.01	150
hydromedusae_sideview_big	0.00	0.00	0.00	76
hydromedusae_solmaris	0.25	0.38	0.30	703
hydromedusae_solmundella	0.44	0.03	0.06	123
hydromedusae_typeD	0.00	0.00	0.00	43
hydromedusae_typeD_bell_and_tentacles	0.33	0.02	0.03	56
hydromedusae_typeE	0.00	0.00	0.00	14
hydromedusae_typeF	0.00	0.00	0.00	61
invertebrate_larvae_other_A	0.00	0.00	0.00	14
invertebrate_larvae_other_B	0.00	0.00	0.00	24
jellies_tentacles	0.20	0.18	0.19	141
polychaete	0.00	0.00	0.00	131
protist_dark_center	0.00	0.00	0.00	108
protist_fuzzy_olive	0.59	0.63	0.61	372
protist_noctiluca	0.26	0.29	0.28	625
protist_other	0.28	0.56	0.37	1172
protist_star	0.41	0.22	0.29	113
pteropod_butterfly	0.22	0.02	0.03	108
pteropod_theco_dev_seq	0.00	0.00	0.00	13
pteropod_triangle	0.00	0.00	0.00	65
radiolarian_chain	0.09	0.03	0.05	287
radiolarian_colony	0.12	0.02	0.03	158
shrimp-like_other	0.00	0.00	0.00	52
shrimp_caridean	0.00	0.00	0.00	49
shrimp_sergestidae	0.07	0.03	0.04	153
shrimp_zoea	0.00	0.00	0.00	174
siphonophore_calycophoran_abylidae	0.00	0.00	0.00	212
siphonophore_calycophoran_rocketship_adult	0.21	0.10	0.14	135
siphonophore_calycophoran_rocketship_young	0.24	0.27	0.25	483
siphonophore_calycophoran_sphaeronectes	0.16	0.20	0.17	179
siphonophore_calycophoran_sphaeronectes_stem	0.00	0.00	0.00	57
siphonophore_calycophoran_sphaeronectes_young	0.22	0.01	0.02	247
siphonophore_other_parts	0.00	0.00	0.00	29
siphonophore_partial	0.00	0.00	0.00	30
siphonophore_physonect	0.07	0.01	0.01	128
siphonophore_physonect_young	0.00	0.00	0.00	21
stomatopod	0.00	0.00	0.00	24
tornaria_acorn_worm_larvae	0.00	0.00	0.00	38
trichodesmium_bowtie	0.26	0.17	0.20	708
trichodesmium_multiple	0.00	0.00	0.00	54
trichodesmium_puff	0.72	0.85	0.78	1979
trichodesmium_tuft	0.21	0.32	0.26	678
trochophore_larvae	0.00	0.00	0.00	29
tunicate_doliolid	0.35	0.03	0.06	439
tunicate_doliolid_nurse	0.11	0.00	0.01	417
tunicate_partial	0.60	0.86	0.71	352
tunicate_salp	0.39	0.47	0.43	236
tunicate_salp_chains	0.00	0.00	0.00	73
unknown_blobs_and_smudges	0.04	0.03	0.03	317
unknown_sticks	0.00	0.00	0.00	175
unknown_unclassified	0.07	0.01	0.01	425
avg / total	0.28	0.34	0.29	30336

References

[1] W. K. Pratt, "Image feature extraction," *Digital Image Processing: PIKS Scientific Inside, Fourth Edition*, pp. 535–577, 1978.

[2] W. Burger, M. J. Burge, M. J. Burge, and M. J. Burge, *Principles of Digital Image Processing*. Springer, 2009.

[3] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, and D. D. Edwards, *Artificial intelligence: a modern approach*, vol. 2. Prentice hall Upper Saddle River, 2003.

[4] M.-K. Hu, "Visual pattern recognition by moment invariants," *IRE transactions on information theory*, vol. 8, no. 2, pp. 179–187, 1962.

[5] I. Basheer and M. Hajmeer, "Artificial neural networks: fundamentals, computing, design, and application," *Journal of microbiological methods*, vol. 43, no. 1, pp. 3–31, 2000.

[6] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*, vol. 1. Springer series in statistics Springer, Berlin, 2001.

[7] C. M. Bishop, *Neural networks for pattern recognition*. Oxford university press, 1995.

[8] P. K. Simpson, *Artificial neural systems: foundations, paradigms, applications, and implementations*. Windcrest/McGraw-Hill, 1991.

[9] D. Pham, "Neural networks in engineering," in *The 9 th International Conference on Applications of Artificial Intelligence in Engineering*, pp. 3–36, 1994.

[10] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech, and time series," *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[11] D. R. G. H. R. Williams and G. Hinton, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

[12] M. Avriel, *Nonlinear programming: analysis and methods*. Courier Corporation, 2003.

[13] B. Ripley, "Neural networks and pattern recognition," *Cambridge University*, 1996.

[14] G. van Rossum and F. L. Drake, *The Python Language Reference Manual*. Network Theory Ltd., 2011.

[15] P. F. Culverhouse, R. Williams, B. Reguera, V. Herry, and S. González-Gil, "Do experts make mistakes? a comparison of human and machine indentification of dinoflagellates," *Marine Ecology Progress Series*, vol. 247, pp. 17–25, 2003.

[16] C. Fiorio and J. Gustedt, "Two linear time union-find strategies for image processing," *Theoretical Computer Science*, vol. 154, no. 2, pp. 165–181, 1996.

[17] R. M. Haralick, K. Shanmugam, *et al.*, "Textural features for image classification," *IEEE Transactions on systems, man, and cybernetics*, no. 6, pp. 610–621, 1973.