Practical Introduction to Neural Networks and Deep Learning: Machine Learning models for Disease Prediction

Aurora Zabot
11907717

Alpen-Adria-Universität — February 2020

Acknowledgements

This work was inspired by [1] and the initial theory from [2].

Abstract

The rapid arise of deep learning techniques in every day problems has aroused the interest in medical applications, in particular linked to visual detection of diseases in early stages. In this work, our focus is modeling an architecture capable of identify if images taken from biopsy reveals a malignant breast melanoma. To do so, we've used to datasets [20, 23], already classified in malignant and benign breast cancer and we've input them, after their splitting in training and test set, into our convolutional model. We've also taken advantages of transfer learning techniques, using the pre-trained architecture DenseNet201 [48], in order to achieve an higher accuracy in the computation. Our results are quite high (around 97% with the first dataset and 93% for the second one), but during the analysis we've also taken into account other metrics related to false alarm, misses and correct detection, to make our work more exhaustive, since in medical field we would prefer having higher false alarm than misses, if there would encounter missclassifications.

1 Introduction

Skin cancer, that is an abnormal growth of skin cells, is one of the most common cancer in USA and worldwide [6]. Its early diagnosis is primarily done visually [4], followed by more deeply analysis if there's the suspect that it could be potentially a melanoma. In this sense, it looks obvious that an early detection is fundamental in order to increase the percentage of outliving (on example, in the aforementioned case the 5-years survival rate is 99%) [6]. To do so, nowadays there are many techniques involved in this process of "early recognition", and one of the most accurate [3] is the implementation of deep neural networks that can classify skin lesions [4]. In particular, in [4] it's highlighted how convolutional neural networks achieves really high level of accuracy in both recognition of general and common cancers and in the early identification of deadliest skin cancer, with results that are comparable with dermatologists' competences.
In our specific work we would focus on *breast cancer*, that is a particular cancer that affects breast tissue

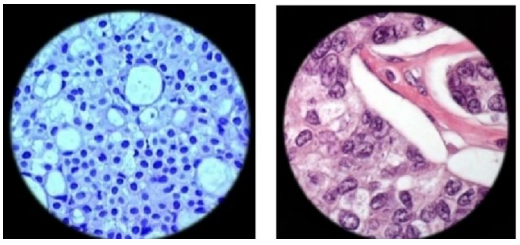

Figure 1: Difference benign (left) and malignant (right) mass, from [9]

[5]. Even though it's not classified as a skin cancer, its formation can be spotted in early stages due to lump in the breasts and scaly patch on the skin [5], that make our work really similar to the one developed in the aforementioned researches. Accordingly to global statistics, about 1 woman on 8 would develops invasive breast cancer and in USA is the second cause of deaths for women compared with all the other kind of tumors [7]. For those reasons, it's understandable why there would be such many researches in this field and how much it's of vital importance to spot in early stage the arise of those tumors.
From a visual point of view, the main difference between a malignant and benign breast cancer is in the shape and the color [8]: in fact, the benign tumors are mostly ovals, whereas the malignant ones have a more irregular profile, wither than the surrounding tissue (see Figure 1); this difference would be fundamental to categorize the different skin lesions from a neural network point of view. Given this distinction, nowadays, many researchers are working in the classification of benign/malignant tumors through mammography images, in order to provide in early stage the more accurate classification possible to recognize the arise of breast tumors [10–16]; those works would be discussed in Section (2).

2 State of Art

There would be many researches to take into account in this field, using different approaches and techniques to correctly classify if a breast tumor is benign or not. Back in 1996, [10] used CNN in order to classify benign and malign masses. To do so, they used segmentation techniques and feature extraction methods (but applied just in small region of interest). The results were around 90% for true-positive and 31% for false-positive. In [11], it has been used principally PCA and discrete wavelet transform (in which wavelets are discretely samples) in order to extract features; in the 98% of the cases, the system correctly classify the lesions. [12] used a multi-scale convolutional neural network to detect masses; they used two different datasets, achieving different results: using DDSM, Digital Database for Screening Mammography, [16], they achieved 90% of accuracy, while using INbreast [17], a dataset specifically designed for this purpose, 85%. The main issue with this work was the really small training and testing set in use (respectively, 39 and 40 cases). In [13], there's the combination of SVM with deep convolutional neural networks, that has achieved an accuracy of 98.44% using INbreast [17]. In [14], using deep convolutional neural networks, the researchers achieved a 89.9% of sensibility in the differentiation between a mass and a normal lesion. They had used also in this case the DDSM [16] dataset. Finally, in [15], we can see the introduction of BCDR-F03, Film Mammography dataset number 3 [18], a new and more complete dataset. Using two different architectures, GoogLeNet and AlexNet, they achieved an accuracy of 88% and 83%.

3 Datasets

We would like to use two different datasets taken from [19] and [22], in order to make our computation more significant. In the following we would analyze how they two datasets are composed.

3.1 Dataset 1 [20]

The BreaKHis dataset [21] contains images taken, in a time period of 12 months in 2014, from microscopic biopsy of both benignant and malignant breast tumors. Each sample is generated through a standard clinical process, whose description goes beyond the purposes of our work; the important thing to grasp in this sense, is that those kind of methods are used to preserve the original structure of the diseased tissue, in order to allow better observations at the microscope and so, more accurate images at the end of the process. Images are RGB (three channel) TrueColor color space(24 bits color depth, 8 bits for each channel) using magnifying factors of 40x, 100x, 200x and 400x (objective 4x, 10x, 20x, 40x). Since the original images acquired had some spurious components (i.e. black corners, text annotations etc), they have been cropped into 700x460 pixels in a portable network graphic (PNG) format, in order to obtain raw images without compromising colors.
The whole dataset is composed by 7909 images; in the table below we would summarize the total distribution of malignant and benign tumors and how they're split.

Magnification	Benign	Malignant	Total
40X	652	1370	1995
100X	644	1437	2081
200X	623	1390	2013
400X	588	1232	1820
Total	2480	5429	7909
Patients	24	58	82

Currently, the database contains four distinct types of benign breast tumors and another four of malignant. The distribution of those different types is summarized once again the the tables below.

Magnification	Adenosis	Fibroadenoma	Phyllodes Tumor	Tubular Adenoma	Total
40X	114	253	109	149	598
100X	113	260	121	150	614
200X	111	264	108	140	594
400X	106	237	115	130	562
Total	444	1014	453	569	2368
Patients	4	10	3	7	24
Magnification	**Ductal Carcinoma**	**Lobular C.**	**Mucinous C.**	**Papillary C.**	**Total**
40X	864	156	205	145	1370
100X	903	170	222	142	1437
200X	869	163	196	135	1390
400X	788	137	169	138	1232
Total	3451	626	792	560	5429
Patients	38	5	9	6	58

In Figure (2) can be seen an image of malignant cancer divided into the four magnification factors (a) 40x, (b) 100x, (c) 200x, (d) 400x. As it's described in [21], the rectangular has been added just for illustrative purposes.

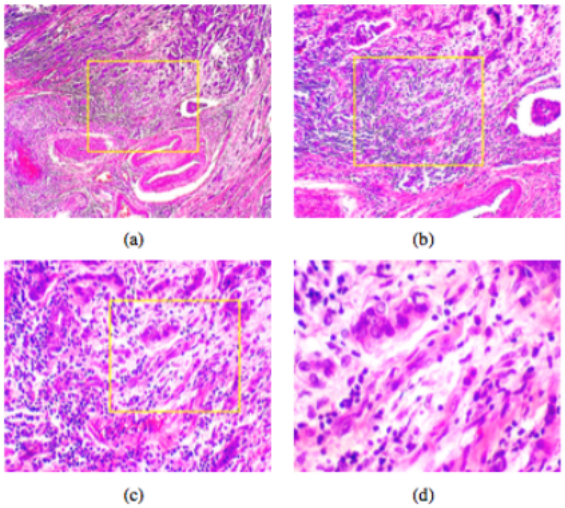

Figure 2: Malignant tumor in different magnification factors from [21]

Finally, each image filename contains information about the method of procedure biopsy, the tumor class, the tumor type, the patient identification and the magnification factor. In a formal way, the notation would be:

```
<BIOPSY_PROCEDURE>_<TUMOR_CLASS>_<TUMOR_TYPE>_<YEAR>-<SLIDE_ID>-<MAGNIFICATION>-<SEQ>

<BIOPSY_PROCEDURE>::=CNB|SOB
```

```
<TUMOR_CLASS>::=M|B
<TUMOR_TYPE>::=<BENIGN_TYPE>|<MALIGNANT_TYPE>
<BENIGN_TYPE>::=A|F|PT|TA
<MALIGNANT_TYPE>::=DC|LC|MC|PC
<YEAR>::=<DIGIT><DIGIT>
<SLIDE_ID>::=<NUMBER><SECTION>
<SEQ>::=<NUMBER>
<MAGNIFICATION>::=40|100|200|400

<NUMBER>::=<NUMBER><DIGIT>|<DIGIT>
<SECTION>::=<SECTION><LETTER>|<LETTER>
<DIGIT>::=0|1|...|9
<LETTER>::=A|B|...|Z
```

3.2 Dataset 2 [23]

Breast Histopathology [23] is a dataset originally formed by 162 slides of Breast Cancer scanned at 40x magnification (0.25 μm/pixel resolution) [24]. The aforementioned images were too large (1010 pixels) to be used in a deep learning work; to solve this problem, each slide has been downsampled to a resolution of 4 μm/pixel, obtaining 277'524 patches of 50x50 (198'738 with benign breast cancer and 78'786 of malignant one), as we can see in Figure (3).

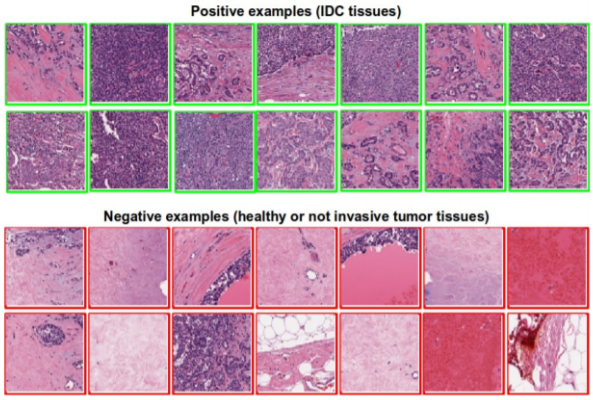

Figure 3: Image patches for positive (malignant) and negative (benign) samples from [24]

We can found each patch filename in the following format:

```
u_xX_yY_classC.png
u := ID patient
(X,Y) := patch were the patch was cropped from
C := class, 0 benignant, 1 malignant
```

4 Data Preprocessing

In this section we would look more deeply into the two datasets we're using. This section would present also a snapshot of the code (since it would be very short), just to make more understandable the preprocessing part of the various datasets.

Libraries

```python
import os
import numpy as np
import matplotlib.pyplot as plt
import cv2
```

```
5
6  from PIL import Image
7  from tqdm import tqdm
8  from keras.utils.np_utils import to_categorical
9  from keras.preprocessing.image import ImageDataGenerator
10 from sklearn.model_selection import train_test_split
```

Here we would import libraries and modules for the preprocessing part. In this sense, we've imported *os*, that from the documentation, this module "provides a portable way of using operating system dependent functionality" [25]. This module has been used to access images in the given directory during the loading part of the code. Then we've imported *numpy* and *matplotlib*, that are respectively the two modules for "scientific computing" [26] and for plotting images and graphs [27]. The first has been used in the whole code, while the second in the plotting part of course. Finally, we would import *cv2*, the opencv library, since for the first dataset we need to resize the images (otherwise the computation becomes unaffordable).
Then we've imported IMAGE from *Pillow* [28], a module that allows load, manipulation and savings of images in many file format; this element has been used in the loading part, in which is needed to import and store the dataset into an array of images.
The module *tqdm* [29] is just a useful tool to display a progress meter during loops, in order to grasp at which point the computation is.
From *Keras* we've imported TO_CATEGORICAL and IMAGEDATAGENERATOR. The first has been used to "convert class vectors into a binary class matrix" [30], as we've done in many computation during lectures, while the second [31] has been used during the data augmentation portion of the code.
Finally, from *SkLearn* we've imported TRAIN_TEST_SPLIT [32], a useful tool to split data into training and test set.

Dataset loading

```
1  def load(directory):
2      image_array = []
3      read = lambda imname: np.asarray(Image.open(imname).convert("RGB"))
4      for imgs in tqdm(os.listdir(directory)):
5          path = os.path.join(directory, imgs)
6          root, figtype = os.path.splitext(path)
7          if figtype == ".png":
8              img = read(path)
9              #img = cv2.resize(img, (210, 135))
10             image_array.append(np.array(img))
11     return image_array
12
13 #Data1
14 benign_train = np.array(load('data1/train/benign'))
15 malign_train = np.array(load('data1/train/malignant'))
16 benign_test = np.array(load('data1/test/benign'))
17 malign_test = np.array(load('data1/test/malignant'))
18
19 #Data2
20 #benign_train = np.array(load('data2/train/benign'))
21 #malign_train = np.array(load('data2/train/malignant'))
22 #benign_test = np.array(load('data2/test/benign'))
23 #malign_test = np.array(load('data2/test/malignant'))
```

Here we define a method to import the dataset and insert it into an array of images. In (2) we've initialized our array of images, then in (3) we're reading multiple RGB images. To do so we've used the module IMAGE. Then in (4), for all the images contained in the directory (given by the path), concatenates the various path components with exactly one directory (i.e. "/") separator (5) [33]. Then, in (6) we split the path name into a pair: root and extension. We're interested in the second portion (the extension one), since we would select all the .PNG elements (7). If the element taken into account is an image, we would select it (8), resize it if it belongs to the first dataset (9) and add it to the array (10). Finally, we would output the final array (11).
Using the aforementioned method, we would load the train and test set. Depending on the dataset we're choosing, we have a different directory of course.

Test and train set

```
benign_train_label = np.zeros(len(benign_train))
malign_train_label = np.ones(len(malign_train))
benign_test_label = np.zeros(len(benign_test))
malign_test_label = np.ones(len(malign_test))

X_train = np.concatenate((benign_train, malign_train), axis = 0)
Y_train = np.concatenate((benign_train_label, malign_train_label), axis = 0)
X_test = np.concatenate((benign_test, malign_test), axis = 0)
Y_test = np.concatenate((benign_test_label, malign_test_label), axis = 0)
```

Here we just simply create the labels, classifying the benignant cancer as 0 and the malignant one as 1. Having the various component, we can merge the data together.

Data shuffle

```
X_shape = X_train.shape[0]
train_shuffle = np.arange(X_shape)
np.random.shuffle(train_shuffle)
X_train = X_train[train_shuffle]
Y_train = Y_train[train_shuffle]

test_shuffle = np.arange(X_test.shape[0])
np.random.shuffle(test_shuffle)

X_test = X_test[test_shuffle]
Y_test = Y_test[test_shuffle]
```

In order not to have all the elements aligned due to concatenate, we would shuffle the data. To do so, we've used NP.ARANGE, that returns evenly spaced values within an interval [34]. Then, we've used NP.RANDOM.SHUFFLE, that only shuffles the array along the first axis of a multi-dimensional array [35]. Finally, we would shuffle both the train and the test sets.

Splitting

```
Y_train = to_categorical(Y_train, num_classes= 2)
Y_test = to_categorical(Y_test, num_classes= 2)

x_train, x_test, y_train, y_test = train_test_split(X_train, Y_train,
                                                    test_size = 0.2,
                                                    random_state = 1)
```

Since we have just a binary classification, we would categorize the train and test sets into two categories, (0, 1) through TO_CATEGORICAL [30]. Then, we would split the two sets using TRAIN_TEST_SPLIT [31]; we would use the common partition of 80% for the train and 20% for the test. We would insert the RANDOM_STATE value in order to make our execution reproducible.

Data augmentation

```
datagen = ImageDataGenerator(zoom_range = 2, rotation_range = 90,
                             horizontal_flip = True, vertical_flip = True)
```

In this process, data augmentation has been done in order to increase the size of training set. We would randomly flip images both horizontally (3rd param) and vertically (4th param) in order to obtain still representative data but from a different point of view. This approach is a common method used to augment the size of the training set if there's not enough samples or if we want to increase the accuracy of our computation, since affine transforms in image categorization are one of the most difficult elements to deal with in image processing.

Image plotting

```
#Data1
width = 30
height = 15

#Data2
```

```
#width = 7
#height = 7

fig = plt.figure(figsize = (width, height))
cols = 4
rows = 3

for i in range(1, cols*rows + 1):
    ax = fig.add_subplot(rows, cols, i)
    if np.argmax(Y_train[i]) == 0:
        ax.title.set_text('Benign')
    else:
        ax.title.set_text('Malignant')
    plt.imshow(x_train[i], interpolation = 'nearest')
plt.show()
```

Here we just display some images and their classifications. Depending on the dataset in use, we would use different sizes in the plot, while the columns and the rows in use are always the same (in our case 3x4). For the total number of plots we want to display (rows*cols), we subplot some images with their appropriate labels (i.e. "Benign" or "Malignant"). Here we would display an example of both the results of this part (see Figures (4), (5)).

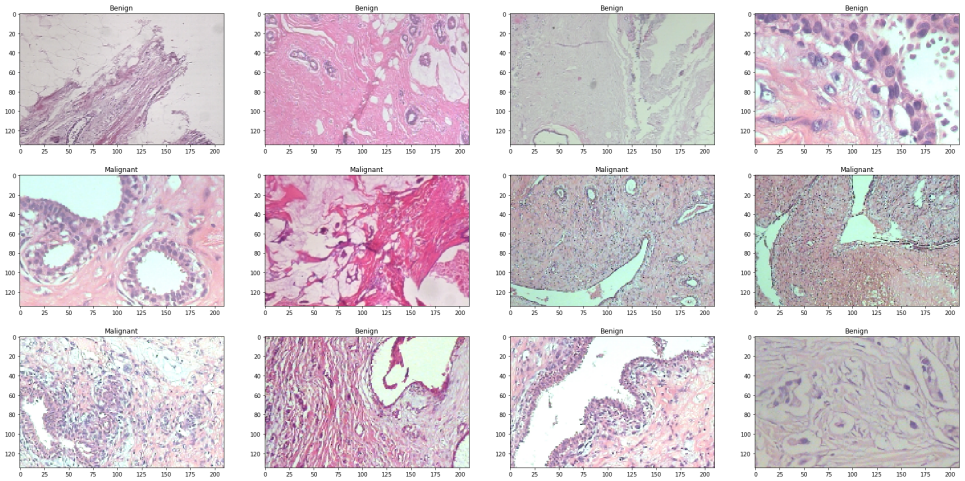

Figure 4: Dataset 1 plot of images and label

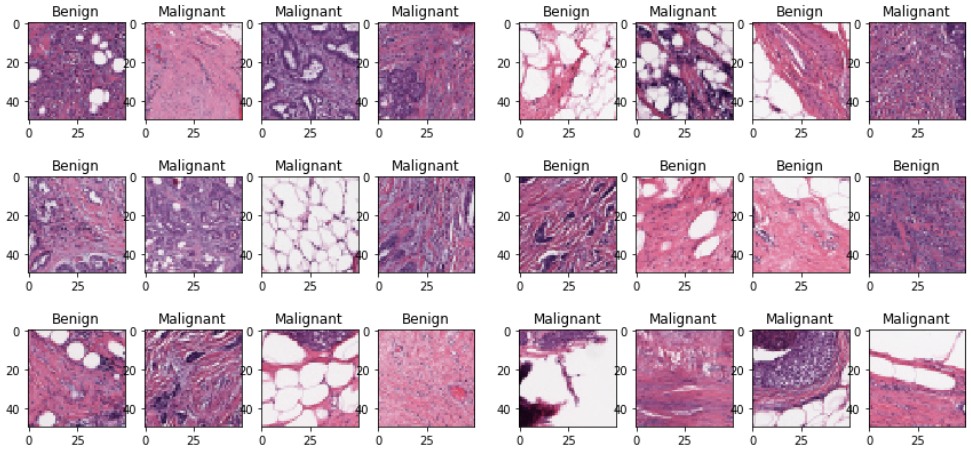

Figure 5: Dataset 2 plot of images and label

5 Problem analysis

After the preprocessing part, we need to take into account which are our goals and how to achieve them. Our main focus is to distinguish between malignant and benign breast cancers just looking at biopsy images; in other words, our problem is related to **Image Classification**. Moreover, since we're dealing with a problem linked to the medical field, we would like achieving an high level of accuracy and we would prefer obtain false alarms instead of misses.
Image Classification is one of the most common tasks in Computer Vision, Machine Learning and Image Processing fields. Formally, the pipeline for this task is quite standard and it can be summarize as follow:

- Our dataset consists of N images, each one with its specific label (in our case 0 or 1)
- The dataset is splitted into the train and the test sets; the first would be used to train our classifier, in order to teach it to correctly match images with classes
- The second set, the test one, is used to verify if the model can correctly classify images. In order to understand the accuracy of the model, we would make a comparison between the match given by the classifier and the real labels

In our particular case, we would use *DenseNet* [48] as a pre-trained model, then a *ResNET*-like architecture [36] in order to correctly match the given images with the correct labels. "ResNET-like" means that we've studied how this CNN architecture is made and we've tried to derive the main features to implement in order to make a consistent model for our task, like the batch normalization, the global average pooling and so on.
To develop an architecture similar to the ResNET one, we've considered the main problems that we can potentially face during the computation, and tried to adapt our model making the choices similar to the ones adopted in [43]. In particular, a very common issue may be overfitting and vanishing of the gradient, or the reduction, due to the deepness of the network, of some metrics at a certain point. To solve the first problems, we have considered to use batch normalization and Adam optimizer. For the second one, instead, we've implemented two callbacks: ModelCheckPoint and ReduceLROnPlateau [37]. The first saves the best performing model, the second adapt the learning rate when metrics stop improving. Using those tricks we have more chances to achieve good results in our computation and increase, in this way, the final accuracy of the whole work.
After this brief overview on our goals and tasks, we're ready to start considering the mathematical model (6) and the algorithm adopted (7).

6 Mathematical formulation

Since our implementation has not too much math to deal with (or in other words, of course there is, but the main elements in the implementation deals with how the architecture is created), we would introduce all the concepts related to convolutional neural networks and pre-trained neural networks. We would use [38–40] for the explanation.

6.1 Pre-trained Neural Networks [38, 40]

During training, the first layers of the network normally identify basic things as lines, geometric shapes and patterns; going deeper and deeper, we start recognizing complex stuff and classify them. This procedure is done for every classifier, no matter about the architecture are you dealing with; so, retrain each time those layers make no sense. Thus, we would use the so called pre-trained architectures (i.e. DenseNet, ResNet etc), whose has been already trained on huge datasets (i.e. ImageNet). Then, we would remove the last layers (where we actually identify the elements needed) and we apply the methods/layers we need for our specific task. Doing so, we would obtain huge benefits in terms of optimal weights, since those pre-trained architectures are run on very huge datasets (i.e. ImageNet has over than 1.4 million of images).

6.2 Convolutional Layer [39, 40]

Convolutional layers are the very basic elements that distinguish a normal neural network to a convolutional one. In pure mathematical terms, convolution can be expressed as:

$$y(t) = (x * h)(t) := \int_{-\infty}^{+\infty} x(\tau)h(t-\tau)d\tau \tag{1}$$

This operation just tells us how the shapes of two different signals can reciprocally modify the other one. Just to convey a common notation, $x(t)$ would be the input signal of our system, while $h(t)$ is the filter convolved with $x(t)$. In a 2D space (since we're dealing with images), this operation can be expressed through "matrices", where the image has the function of $x(t)$ and the so called *kernel* (or feature detector) has the function of $h(t)$. The shape of the kernel determines how many pixels are involved in the convolution operation, while the values define the kind of operation we're applying on the image (on example, convolving an image through a box filter would blur it, and in linear domain it would be the same of passing a signal through a low pass filter). Using a sliding-window approach (i.e. the kernel is passed throughout the whole image shifting the matrix pixel-by-pixel), we would make a dot product by each pixel and the corresponding value in the kernel, performing at the end the convolution operation. If the kernel shape is not suitable w.r.t. the image dimensions (i.e. the height and width of the image is not divisible for the ones of the kernel), we would perform padding. It's important to highlight that this operation doesn't change the shape of the image, differently from the pooling operation. Those filtering operations are the fundamental steps to grasp all the informations needed for the feature extraction and detection used for the final classification of the image. This function is perfectly expressed by Figure (6)

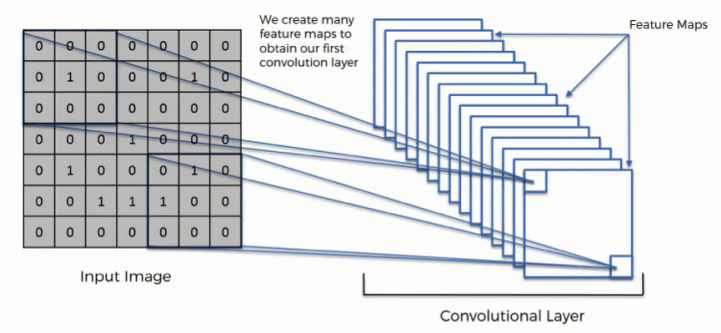

Figure 6: Convolutional operation from [39]

6.3 ReLU [39]

The ReLU activation function has the main aim to "increase the non-linearity of our function" [39]. Images

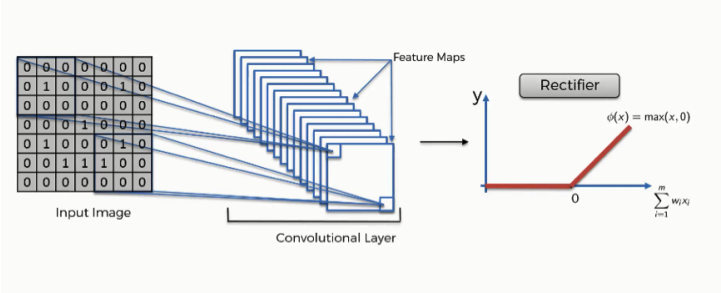

Figure 7: ReLU from [39]

are not linear by their own nature, since they contain features as corners, borders, drastic change in colors, edges and so on. Applying ReLU, we further increase the non-linearity of the image before convolutional operations, in order to grasp more complex relationships in the data to be learned. Moreover, ReLU avoid

saturation during backpropagation. This activation function can be figure out thanks to Figure (7). From a mathematical point of view, the ReLU can be expressed as:

$$f(x) = max(0, x) = \begin{cases} x & if x > 0 \\ 0 & if x \leq 0 \end{cases} \tag{2}$$

6.4 Pooling Layer [39, 40]

The pooling opearation concerned with the so called "spatial variance", that means that given different images with the same subject, we can recognize that, despite all the differences (textures, angles, luminance condition and so on), it's always the same object. There are many types of pooling, but the most common are *average, max* and *sum pooling*. Pooling, in fact, uses a sliding-window approach with a kernel as well, but not doing a convolutional operation: from the values inside the window, depending on the type of pooling operation in use, we would choose just the average/max/sum of the values, operating at the end a downsampling operation. Differently from the convolutional layer, indeed, we would change the shape of the output in this phase, reducing it depending on the size of the kernel in use. With this operation, we would drop unnecessary informations for our feature computation, making the job of the network more efficient from this point on. All those stuff not only are used to grasp the spatial variance, as already said, but also to prevent overfitting due to the complexity of the model. Mathematically, we can express the pooling operation with a nxn kernel as follow (we just compute it for one single window for simplicity, but we need just to iterate the operation on the whole matrix):

$$\begin{cases} \text{Max pooling} & max_{i \in (0, n-1)}\{x_i\} \\ \text{Average pooling} & \frac{1}{n}\sum_{i=0}^{n-1} x_i \\ \text{Sum pooling} & \sum_{i=0}^{n-1} x_i \end{cases} \tag{3}$$

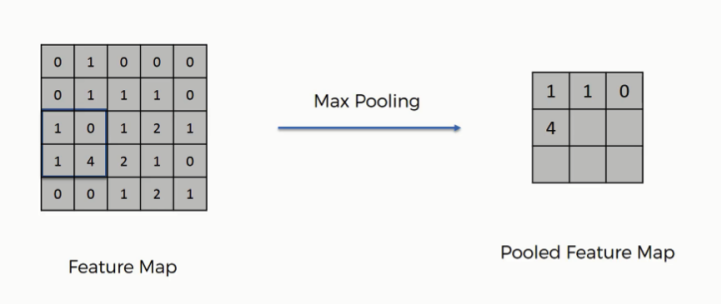

Figure 8: Example of max pooling operation from [39]

6.5 Dropout [42]

Dropout is a regularization method that has the main aim of reducing overfitting, allow the network not to count too much on single nodes and improve generalization error in all kinds of neural networks. The procedure is really simple and computational efficient, since consists just in randomly dropping out (with an arbitrary rate) nodes during training. In very large networks, this technique is required also for light up the computational complexity of the training process. It can be used almost in every kind of hidden layer, but not in the output one. Finally, as a remark, this approach is really different from using a smaller dataset, that at a first sight could be a good alternative approach: in fact, in the first case we won't lose all the features learned during the implementation, whereas in the second case we are not even sure that all the aforementioned features would be learned. The mathematical formulation is just a random drop of the elements contained in the layers.

6.6 Flattening [39, 40]

Flattening is a quite simple operation, since we just "flatten" all the pooled feature map obtained in order to feed the fully connected layer, as can be seen in Figure (9). Mathematically it can be described as:

$$\begin{cases} i \in [0, \text{rows x cols} - 1] = [0, I] & \text{(rows, cols)} = (M, N) \\ f_i = M_{m,n} & \text{for m = 0 to M, n = 0 to N} \end{cases} \tag{4}$$

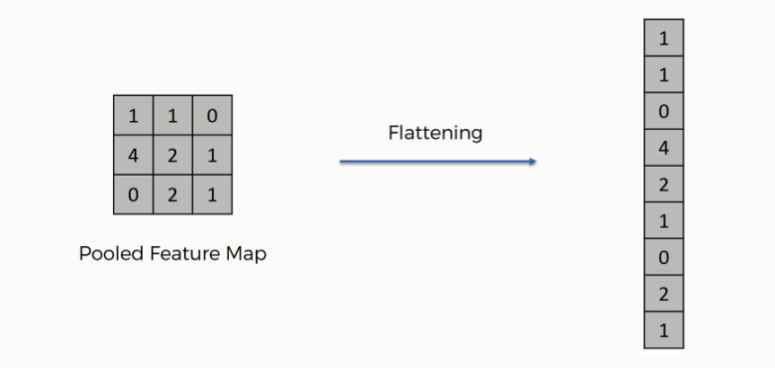

Figure 9: Flattening operation from [39]

6.7 Fully Connected Layer [39, 40]

As mentioned before, the fully connected layers received in input the flattening vector of features, as exposed in Figure (10), and just at this point there's an high degree of accuracy in the classification of the classes.F To be more accurate, we would further go into computation through fully connected layers.

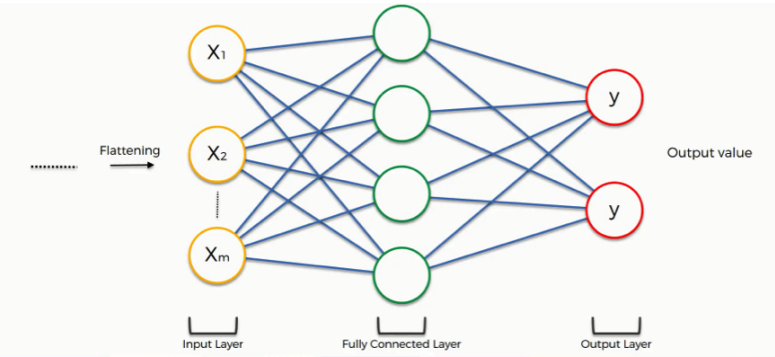

Figure 10: Fully connected layer from [39]

Differently from the other layers, the fully connected ones have the particularity that each neuron is connected to all the nodes of the next layer. In this way, each feature detected is communicate to the next layer: all the classes can check if this information is relevant or not for its identification and they can decide consequently. This operation is taken several times during the training, and each time the accuracy in the identification of the features characterizing the subject increase. We can't give a mathematical background in a few about how a fully connected network is composed, but we can summarize it in Figure (11)

7 Formal approach

Since our task deal with image classification using DenseNet and additionally some features architecture similar to the ResNET one, we would firstly introduce how ResNET and DenseNet work and then how we've derived the elements for our implementation.

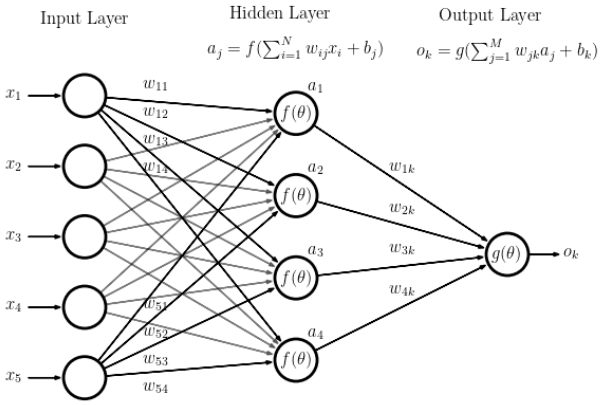

Figure 11: Fully connected layer architecture [41]

7.1 ResNET basics

From what we can learn from [43], using deep neural networks for image classification "led to a series of breakthroughs". In particular, those problems are related to the vanishing/exploding of the gradient, that may be solved using normalization approaches, and the more serious problem of *degradation*. When the network's depth start increasing, at a certain point the accuracy may saturate and start degrading rapidly. This term, as it's highlighted in the paper [43], underlined that "not all systems are similarly easy to optimize". To solve this point, the authors introduces the so called **deep residual learning**, that can be exploited through Figure (12).

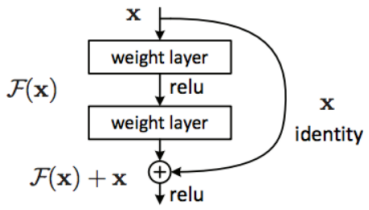

Figure 12: Residual learning block from [43]

The architecture introduces two types of "shortcut connections" [44]: the identity and the projection shortcut. Depending on the dimension of the input/output layer, it uses, respectively, the identity shortcut if the dimension of the input is the same of the output, otherwise it uses the projection one. Suppose we want to learn $y = H(x)$, as it's normally done. We can directly learn this or use an auxiliary function $F(x)$ defined as $F(x) = H(x) - x$ (*residual mapping*), obtaining the output $y = F(x) + x$. A building block in this architecture, so, can be defined as $y = F(x_i, W_i) + x$, where $F(\cdot)$ can include also multiple layers (i.e. in Figure (12) it's $F = W_2[ReLU(W_1x)]$). The case just described doesn't introduces any new parameter in the network; in fact, we're considering the *identity shortcut*, that is used, as already said, when the dimensions of the input and the output are the same. On the other hand, if it's not the case, i.e. the two dimensions are different, we need to adopt another approach (*projection shortcut*). To do so, we can consider two different approaches: the involvement of the projection matrix W_s or a simple zero-padding. In the first case, the main task of the projection matrix is to project x in the same space of $F(x)$. The output in this sense would be $y = F(x_i, W_i) + W_sx_i$, that, differently to the previous case, introduces new parameters. In the second case, instead, we just zero-pad x until it has the same dimension of the output, introducing no new parameters. This is the main feature that characterize this architecture. All this introduction has been made in order to clarify how DenseNet actually works, since we've chosen it as a pre-trained model.
For the implementation point of view [43, 44], that it's our main focus in this architecture, ResNET performs:

- Data Augmentation: flipping images to increase the dataset size
- Batch normalization

- SGD with momentum: with minibatch of size 256
- Momentum: 0.9
- Learning rate: start from 0.1 and decrease of a factor 10 due to
- Weight decay: 0.0001
- Global average pooling and no dropout

All those things have been elaborated in order to construct the model we've used in our implementation, that would be exposed in section (7.3).

7.2 DenseNet [48]

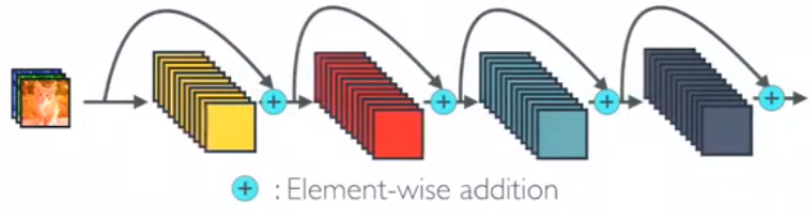

Figure 13: ResNET concept from [48]

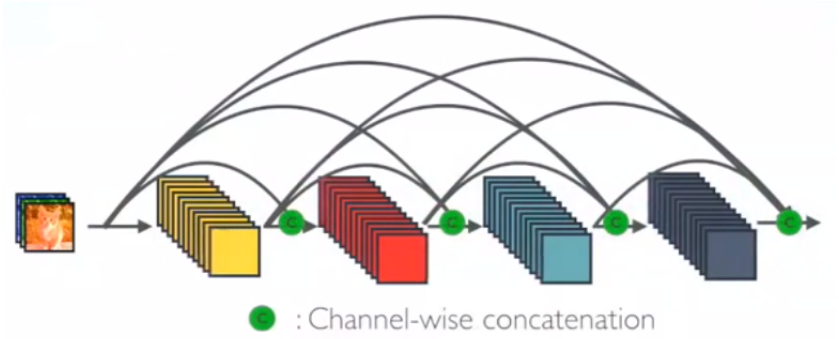

Figure 14: DenseNet concept from [48]

All the introduction done in the previous chapter would be useful to understand how DenseNet works. As we can see from Figure (13) and (14), the trick used in DenseNet is similar to the one offered by ResNET, but more articulate: "each layer obtains additional inputs from all preceding layers and passes on its own feature-maps to all subsequent layers" [48]. In other words, we're concatenating all the "collective knowledge" from the previous layers. Receiving all the previous knowledge, the whole network can be more compact, increasing both memory and efficiency.
From an architecture viewpoint, for each layer we have ReLU and 3x3 kernel size convolution; furthermore, we have the so called Pre-Activation Batch Norm (BN), that is just a normalization before the activation function, that from an implementation point of view seems to work better than the version exposed in the official paper [49]. Due to complexity, it is performed BN-ReLU-1×1 convolution and then BN-ReLU-3×3 convolution. Between two dense blocks, there would be 1x1 convolution and 2x2 average pooling. All the feature outputs have the same size, in order to be better concatenate. At the very end of the network, is done global average pooling and there is a softmax classifier.
There are many advantages due to this architecture, that can be summarize as follow:

- **Strong gradient flow**: the error signal, due to the architecture, can be pushed to the first layers more directly (deep supervision)

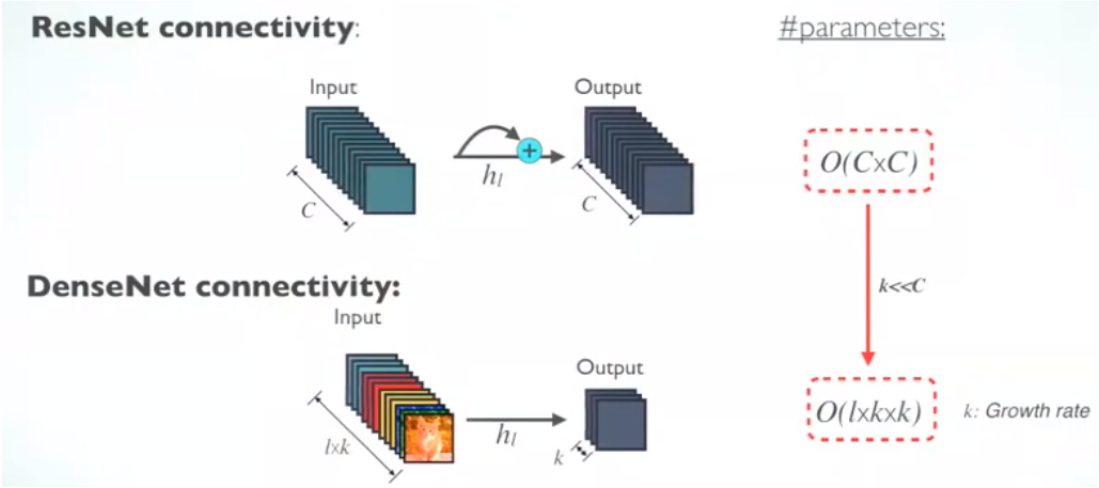

Figure 15: Comparison number of parameters ResNET and DenseNet, from [48]

- **Number of parameters and efficiency**: looking at Figure (15), for ResNET, the number of parameters is directly proportional to CxC, whereas the ones of DenseNet are lxkxk; thus, DenseNet is much smaller than ResNET
- **Diversification of the features**: due to the architecture, we would enrich a lot the patterns found during feature extraction
- **Low complexity features**: in DenseNet, classifier uses features of all complexity levels, giving smoother decision boundaries

Now that we have an entire overview on the models we've used, we can describe our architecture.

7.3 Our model

As we've already said, we've used **DenseNet201** [47, 48] as pretrained weights, following the suggestion of the quoted article. The complete description of the model is given in (7.2).
By now, we've already explained and show how we've done data augmentation. In particular, as it's done in the ResNET architecture, we've flipped our images in vertical and horizontal axis. This technique is one of the easiest way to increase the size of the training set and there are many statistics [45] that proves its usefulness on dataset as CIFAR-10 and ImageNet.
We've used also batch normalization before the final dense layer of two neurons for the two output classes (we've used softmax as activation function).
Instead of using SGD with momentum, we've chosen Adam optimizer, since is probably one of the best optimizer at the moment, with a learning rate of 10^{-4}. We would use a batch of different size depending on the kind of dataset we're dealing with (but normally quite small). The choice has been made considering that large batch sizes surely converge to optimality, but in a really slow way, not leading to generalization. On the other hand, it has been shown that using small batch sizes the network trains more faster and achieves really good results, but it's not ensure to achieve optimality [46]. To highlight this fact, we've run many tries with different batch dimensions.
We've done global average pooling as well, but differently from ResNET, we've also made dropout of 50% to reduce overfitting.
In order to simulate the adaptation of the learning rate proposed in ResNET, we've used ReduceLROnPlateau, as it has already been explained, in order to reduce the learning rate when a metric stops improving [37].
Finally, we recall that we've also introduced ModelCheckPoint in order to save the best performing model.

7.4 Algorithm

```
def architecture():
    set dimension of images as (width, height, channels)
    set pretrained weights on 'imagenet' using DenseNet (input (dimensions))

    start building the model:
            add the pretrained model
            do global average pooling (2D, for images)
            do 50% of dropout
            do batch normalization
            add two dense layers using softmax as activation function

    compute the loss through Adam optimizer (on accuracy metrics)
        with a learning rate of 0.0001

    return the architecture

Create the model = architecture()
Return an overview on the elements of the network

Reduce learning rate if metrics do not increase (ReduceLROnPlateau)
Create checkpoint on best model evaluation (ModelCheckpoint)

Train the model on the training set

Plot loss and accuracy

Make predictions on test set
Create confusion matrix (evaluate missclassifications)
Report on metrics prediction

Example plot on predicted and actual results
```

7.5 Architecture

The main passages of the project are summarize in the following flowchart (16); we didn't express all the passages in order to be more light, stressing out just just the main differences in the implementation of the two datasets.

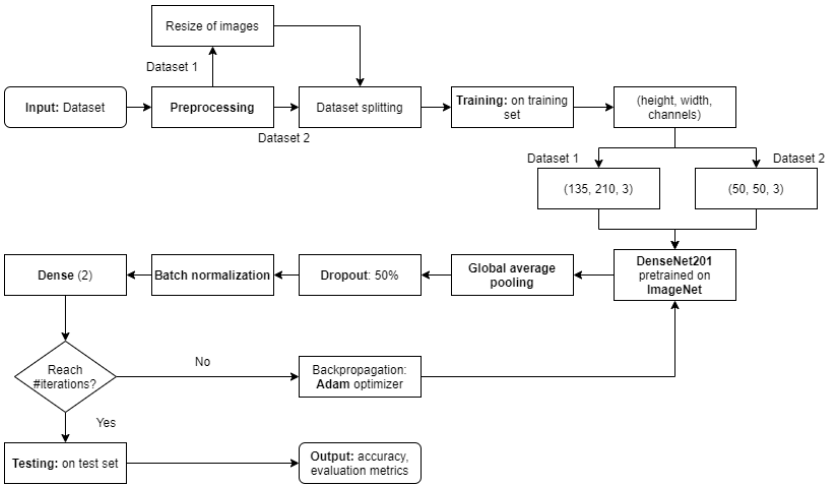

Figure 16: Project Architecture

Finally, in (17) we have printed the parameters of the network in DenseNet, respectively of the first and the second datasets.

Layer (type)	Output Shape	Param #
densenet201 (Model)	(None, 4, 6, 1920)	18321984
global_average_pooling2d_1 (	(None, 1920)	0
dropout_1 (Dropout)	(None, 1920)	0
batch_normalization_1 (Batch	(None, 1920)	7680
dense_1 (Dense)	(None, 2)	3842

Total params: 18,333,506
Trainable params: 18,100,610
Non-trainable params: 232,896

Layer (type)	Output Shape	Param #
densenet201 (Model)	(None, 1, 1, 1920)	18321984
global_average_pooling2d_1 (	(None, 1920)	0
dropout_1 (Dropout)	(None, 1920)	0
batch_normalization_1 (Batch	(None, 1920)	7680
dense_1 (Dense)	(None, 2)	3842

Total params: 18,333,506
Trainable params: 18,100,610
Non-trainable params: 232,896

Figure 17: Parameters respectively from Dataset 1 and Dataset 2

8 Implementation

Architecture

```
def model_architecture():
    #Dataset 1
    #height = 135
    #width = 210
    #channels = 3

    #Dataset 2
    height = 50
    width = 50
    channels = 3

    spine = DenseNet201(weights = 'imagenet', include_top = False,
                        input_shape = (height, width, channels))

    model = Sequential()
    model.add(spine)
    model.add(layers.GlobalAveragePooling2D())
    model.add(layers.Dropout(0.5))
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(2, activation = 'softmax'))

    learning_rate = 1e-4
    model.compile(loss = 'binary_crossentropy', optimizer =
                  Adam(lr = learning_rate), metrics = ['accuracy'])

    return model

model = model_architecture()
```

Here we would define a function to create our model. Depending on the dataset in use, we would use, we would have different sizes of our images (from (2) to (10)). Then, as mentioned in 7.3, we would use the pre-trained model DenseNet201 (12), whose weights have been trained on ImageNet. On top of the architecture we import the pre-trained model (16), then we made global average pooling (17) in order to prevent overfitting and for increase the spatial variance. In (17) we introduced a dropout of 50%, value commonly used, to prevent overfitting as well and improve generalizations. Then, in (18) we've done batch normalization to prevent the problem of the vanishing of the gradient. At the end, in (19), we set two dense layers and a softmax activation function for binary classification. For the optimization, we've chosen Adam (23), since currently is one of the best optimizers. We've chosen a learning rate of 0.0001, also in this case using a common value (22). Finally, we would return the output of our architecture (26). For sake of clarity, we output a short summary of the model (28).

Callbacks

```
reduce_larning_rate = ReduceLROnPlateau(monitor = 'val_accuracy', factor = 0.2,
                                        patience = 5, verbose = 1, min_lr = 1e-7)

path = 'checkpoint.hdf5'
checkpointer = ModelCheckpoint(path, monitor = 'val_accuracy', verbose = 1,
                               save_best_only = True, mode = 'max')
```

Implementation of the two callbacks function, that have been already mentioned in the previous sections. In a few words, callbacks are really useful tools used to check internal states during training. In (1) we're implementing ReduceLROnPlateau, that is a useful tool to reduce the learning rate when a metric stops growing. In our case, we look at the accuracy parameter: if the accuracy stops increasing, we would reduce of 0.2 the new learning rate. We would wait 5 epochs before decreasing the learning rate if the metrics stop improving (that is the "patience" parameter). Moreover, we set 0.0000001 as the lower threshold. In (5), on the other hand, we would implement ModelCheckpointer, that saves the model after every epoch: if the metrics increase it substitutes the old file, otherwise it would keep the best model achieved until that time; in our case, we would save it in "filepath" (4). We would monitor the accuracy as well and we set to save just the model that maximizes the accuracy.

Confusion Matrix

```
def plot_confusion_matrix(cm, classes, title, colormap, normalize):
    plt.imshow(cm, interpolation = 'nearest', cmap = colormap)
    plt.title(title)
    plt.colorbar()

    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes)
    plt.yticks(tick_marks, classes)

    fmt = 'd'
    thresh = cm.max() / 2
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt), horizontalalignment = 'center',
                 color = 'white' if cm[i, j] > thresh else 'black')

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

confusion_mx = confusion_matrix(Y_true, Y_pred)
title = 'Confusion matrix for Skin Cancer'
labels = ['Benig, 0', 'Malignant, 1']
cmap = plt.cm.Reds
normalization = False

plot_confusion_matrix(confusion_mx, labels, title, cmap, normalization)
print('Confusion matrix for Skin Cancer, without normalization: \n', confusion_mx)
```

Confusion matrices are used to evaluate the quality of the output of a classifier [50]. In the diagonal we would find the number of points for which the predicted label is equal to the true label, whereas the off-diagonal elements are the miss-classifications. In our case, we want also to plot the confusion matrix for sake of clarity, following the tutorial found in [51]. All the passages are quite mechanical and are well described in the code. In a few words, we would divide our confusion matrix plot into four sub-blocks (6) after the computation of the confusion matrix from sklearn (20). Then we would define the labels in (7) and (8) and we would place them in the image. From (10) to (14), we would create the plot defining that the more that the colors are intense, the higher is the number of elements in that sub-block. The labels are defined in (22), while the color mapping in (23). We are not interest in normalization as well (24). Then we would output both the plotted confusion matrix (26) and the confusion matrix offered by sklearn (27).

Plot of the results

```
def classification(code):
    if code == 0:
        return 'Benign'
    else:
        return 'Malignant'

fig = plt.figure(figsize = (width, height))
cols = 4
rows = 3

```

```
for i in range(len(correct)):
    ax = fig.add_subplot(rows, cols, i + 1)
    predicted_corr = np.argmax(Y_pred_tta[correct[i]])
    actual = np.argmax(Y_test[correct[i]])
    ax.set_title('Predicted result: ' + classification(predicted_corr) + '\n' +
                 'Actual result: ' + classification(actual))
    plt.imshow(X_test[correct[i]], interpolation = 'nearest')
plt.show()

fig = plt.figure(figsize = (width, height))
cols = 4
rows = 3

for j in range(len(miss)):
    ax = fig.add_subplot(rows, cols, j + 1)
    predicted_miss = np.argmax(Y_pred_tta[miss[j]])
    actual = np.argmax(Y_test[miss[j]])
    ax.set_title('Predicted result: ' + classification(predicted_miss) + '\n' +
                 'Actual result: ' + classification(actual))
    plt.imshow(X_test[miss[j]], interpolation = 'nearest')
plt.show()
```

Finally we would plot some results of our implementation, in order to see graphically which images are more difficult to classify and which are not. In (1) we just define a function to label the plots, in order to be more clear. Then, in (11) we would plot the correct matches, while in (24) the miss-classifications. The code is quite simple and self explanatory, but it has been more commented in the code in any case. Also in this case, depending on the size of the images we would just changing the dimension of the plot.

9 Comments

Here we just want to give a brief overview using bullets on what we learned during the implementation of the code, without any consideration on the results, that would have its own Section 11.

- The whole implementation takes a lot of time to train, not having the possibility of using the parallelism of GPUs for both datasets. In particular, we've found that the first dataset would take around 1 hour and half to train, whereas the second one at least 30 minutes.
- Using huge datasets it's not forbidden, but actually would lead to really long times and not always give back a good evaluation (i.e. on the second dataset I had an accuracy that became poorer using more samples).
- The final metrics really depends on the dimension of the batch. This would be clear looking at the results I had during the various sessions of training (you can find them in the attached zip file).
- Focus on the first dataset: it occupies too much memory (OOM error) if runned without resize/with too high batch. Without resize the images, the computation starts but becomes unaffordable from a time complexity point of view.
- From the third epochs on: huge improvements on the accuracy.

10 Performance evaluation

In order to evaluate our performaces, we would use tools as **accuracy, precision, recall, F1-score** and **confusion matrix**. All those stuff are given by ACCURACY_SCORE [52], CLASSIFICATION_REPORT [53] and CONFUSION_MATRIX [50], functions offered by sklearn. Accuracy is one of the most common metrics to evaluate the performances of a model, but actually taken by itself, give us nothing in terms of misses and false positive. So, in order to have a more precise idea about those metrics, that in medical field are fundamental, we would take into account precision, recall and F1-score. The first defines the ratio between the true positives and the sum of true and false positives, so, it indicates the percentage of the correct positives. The second one, indicates the ratio between true positives to the sum of true positives and false negatives; in other words, it evaluates the percentage of correct classification of true positives. Whereas the last one is the weighted average of precision and recall. In the following we would output

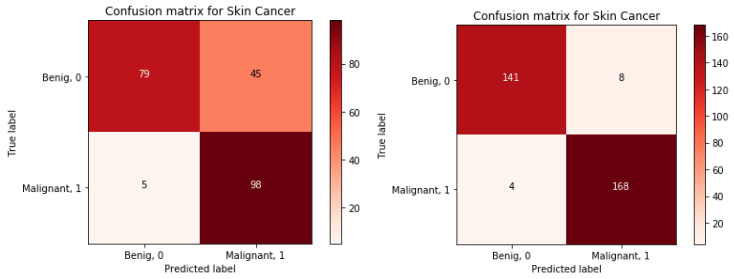

Figure 18: Dataset 1 and Dataset 2 - Correct classifications and Miss-classifications

some of the results of the aforementioned parameters. The concept of confusion matrix, has already been treated in Section (8), and can be summarize as a tool to evaluate the quality of our computation in terms of detection (diagonal), misses (bottom left) and false alarm (top right); the plot of confusion matrices can be visualized in Figure (18).

10.1 Dataset 1

```
-----------------------------------------------------
Accuracy achieved: 0.9693251533742331
-----------------------------------------------------
Confusion matrix for Skin Cancer, without normalization:
[79 45]
[ 5 98]
-----------------------------------------------------

-------------------------------------
      precision    recall  f1-score
-------------------------------------
0       0.96      0.55      0.70
-------------------------------------
1       0.64      0.97      0.77
-------------------------------------
```

10.2 Dataset 2

```
-----------------------------------------------------
Accuracy achieved: 0.9162303664921466
-----------------------------------------------------
Confusion matrix for Skin Cancer, without normalization:
[141   8]
[  4 168]
-----------------------------------------------------

-------------------------------------
      precision    recall  f1-score
-------------------------------------
0       0.93      0.99      0.96
-------------------------------------
1       0.99      0.94      0.96
-------------------------------------
```

11 Results

This section is strictly related to the previous one 10. Now we would provide some plots and comments about the results obtained and how should we work to improve our model.

11.1 Dataset 1

Looking at the results obtained on the first dataset, we would see that the total accuracy is higher than the second one (96.93%), but actually the percentage of false positive is high as well. Since we're in medical field, we would prefer having false positive instead of misses, but actually we would like to achieve a level of accuracy slightly higher with less false alarm/misses. There would be many reasons due to this result; in my opinion, the main issue is related to the fact that our training set was too little to achieve better results. Even if we've tried to have the various training images possible, the final training set had just around 400 images for malignant cancer and the same for benign one. By the way, we had to take a trade-off: due to the architecture of our computer, we had to choose if making the training really long (more than a day) and use a higher set or, on the other hand, making the training set smaller and make the computation affordable. Moreover, this dataset had images whose dimensions were quite huge (450x700). At the beginning we've tried to use them without resizing. This gave us error related to the occupation of the memory. Combining the resize of the images with the reduction of the datasets, we decrease a lot the computational complexity, but degrading the final result as well. Changing many times the batch size, we've seen that in this dataset this metrics affects a lot the final computation (the different batch sizes were 10, 15, 20). In particular, we've seen that using a greater batch we would improve a little the computation, but at cost of very slow training, whereas on the other hand, a smaller batch never lead us OOM, but it achieved worse results than with the higher batch.

Looking more deeply image (19), we can see that the large majority of miss-classifications has been

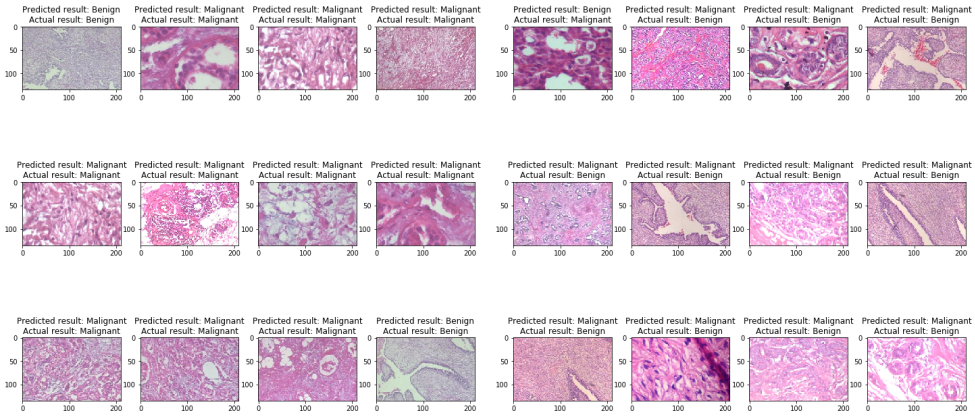

Figure 19: Dataset 1 - Correct classifications and Miss-classifications

done mostly on images where the tumor there "elongated". This is something that I've seen also in the other results (that can be found in the attached zip file). Probably, the best way to deal with this miss-classification was just to take more images with this "shape" in order to make a better classification on them.

11.2 Dataset 2

The second dataset actually achieved a lower accuracy (91.62%), but actually the false alarm and misses were very low. In fact, looking at the f1-score, we can see that is really high (the best is the model, the higher is the score). In this case we used a greater training set, more various, and this probably make the final computation better. Moreover, the images were quite small (50x50), and we've seen that the training took a very little time compared to the first dataset. Also in this case we've tested the set with different batch sizes (10, 23). In this case, the final result was really affected by the dimension of the batch, achieving better results as higher the batch was. Looking at the shape of the miss-classifications, we can see that most of them are or really "homogeneous", or really "blubbled". As we've said before, a way

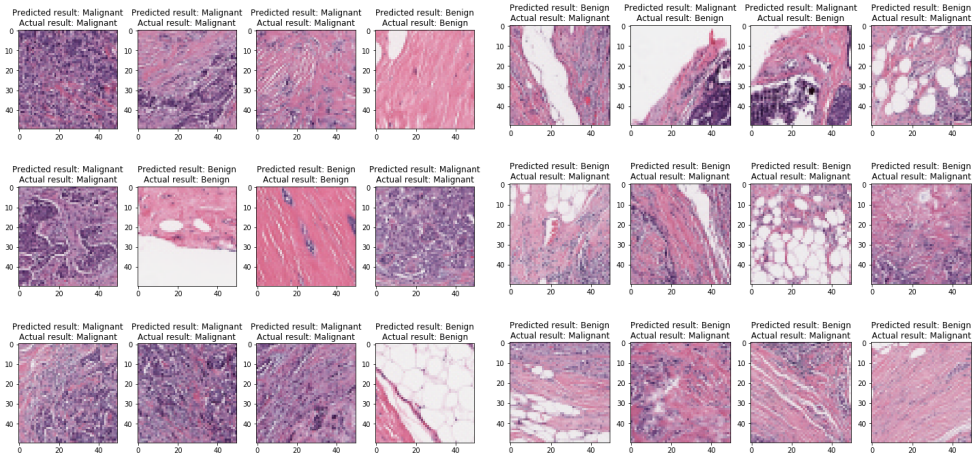

Figure 20: Dataset 2 - Correct classifications and Miss-classifications

to correct those stuff is to introduce more samples with this shape, in order to augment the probabilities that also those features can be learned by the classifier.

11.3 Improvements

Here we've collected some ideas to further improve our work; we just summarize them through bullets in order just to give a brief sketch

- In order to improve a bit the computation, we should enrich the training set, introducing more various samples to make the training more accurate.
- In [54] it's suggested to use progressive resizing in order to achieve better performances in this field
- In [55] it's suggested as well to use both convolutional neural networks and segmentation in order to obtain higher results in the computation

12 Conclusions

In this work we've demonstrated how to classify benign and malignant breast cancer using images taken from biopsy. The effectiveness of convolutional neural networks and of transfer learning is evident at this point: potentially without any help of a specialist, we would possibly more or less classify, just taking a photo, if a lesion may be dangerous or not; in this sense, even thought this project is really far to be a good model, it also highlights how much successful can be the introduction of deep learning in everyday problems.

References

[1] *Shirin's playgRound*: Building meaningful machine learning models for disease prediction

[2] *Dr Shirin Glander, Dep. of Genetic Epidemiology, Institute of Human Genetics, University of Münster*: Building meaningful machine learning models for disease prediction slides

[3] *Science Daily*: Man against machine: AI is better than dermatologists at diagnosing skin cancer

[4] *Nature*: Dermatologist-level classification of skin cancer with deep neural networks

[5] *Wikipedia*: Breast cancer

[6] *Skin cancer foundation*: Skin Cancer Facts & Statistics

[7] *BreastCancer.org*: U.S. Breast Cancer Statistics

[8] *PeerJ - the Journal of Life and Environmental Sciences*: Breast cancer detection using deep convolutional neural networks and support vector machines

[9] *Khelassi Abdeldjalil, 2014*: Explanation-aware computing of the prognosis for breast cancer supported by IK-DCBRC: Technical innovation

[10] *B. Sahiner, Heang-Ping Chan, N. Petrick, Datong Wei, M.A. Helvie, D.D. Adler, M.M. Goodsitt, 1996*: Classification of mass and normal breast tissue: a convolution neural network classifier with spatial domain and texture images

[11] *Maha Sharkas, Mohamed Al-Sharkawy, Dina Ahmed Ragab, 2011*: Detection of Microcalcifications in Mammograms Using Support Vector Machine

[12] *Neeraj Dhungel, Gustavo Carneiro, Andrew P. Bradley, 2015*: Automated Mass Detection in Mammograms Using Cascaded Deep Learning and Random Forests

[13] *Itsara Wichakam, Peerapon Vateekul, 2016*: Combining deep convolutional networks and SVMs for mass detection on digital mammograms

[14] *Shintaro Suzuki, Xiaoyong Zhang, Noriyasu Homma, Kei Ichiji, Norihiro Sugita, Yusuke Kawasumi, Tadashi Ishibashi, Makoto Yoshizawa, 2016*: Mass detection using deep convolutional neural network for mammographic computer-aided diagnosis

[15] *Fan Jiang, Hui Liu, Shaode Yu, Yaoqin Xie, 2017*: Breast mass lesion classification in mammograms by transfer learning

[16] *University of Florida, DDCM* DDSM: Digital Database for Screening Mammography

[17] *Breast Research Group* INbreast

[18] *Breast Cancer Digital Repository* BCDR

[19] *Laboratório Visão Robótica e Imagem*: Medical Images

[20] *Laboratório Visão Robótica e Imagem*: Breast Cancer Histopathological Database (BreakHis)

[21] *Fabio A. Spanhol, Luiz S. Oliveira, Caroline Petitjean, Laurent Heutte, 2016*: A Dataset for Breast Cancer Histopathological Image Classification

[22] *Kaggle*: Datasets

[23] *Paul Mooney, Kaggle*: Breast Histopathology Images

[24] *Angel Cruz-Roaa, Ajay Basavanhallyb, Fabio Gonzáleza, Hannah Gilmorec, Michael Feldmand, Shridar Ganesane, Natalie Shihd, John Tomaszewskif, Anant Madabhushi, 2014*: Automatic detection of invasive ductal carcinoma in whole slide images with Convolutional Neural Networks

[25] *Phyton documentation*: Os Module

[26] *NumPy.org*: NumPy

[27] *matplotlib.org*: Matplotlib

[28] *Pillow (PIL Fork)*: Pillow

[29] *tqdm documentation*: tqdm

[30] *Keras Documentation*: To Categorical

[31] *Keras Documentation*: ImageDataGenerator class

[32] *Scikit-Learn*: Train Test Split

[33] *Geek for Geeks*: Python - os.path method

[34] *SciPy.org*: numpy.arange

[35] *SciPy.org*: numpy.random.shuffle

[36] *Medium*: Understanding and Implementing Architectures of ResNet and ResNeXt for state-of-the-art Image Classification

[37] *Keras Documentation*: ModelCheckpoint, ReduceLROnPlateau

[38] *Toward Data Science*: How do pre-trained models work?

[39] *Super Data Science*: Convolutional Neural Networks (Step 1 to 4)

[40] *Practical Introduction to Neural Networks and Deep Learning*: Slides on CNN and Transfer Learning with CNNs

[41] *AstroML*: Neural Network Diagram

[42] *Machine Learning Mastery*: A Gentle Introduction to Dropout for Regularizing Deep Neural Networks

[43] *Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, 2015*: Deep Residual Learning for Image Recognition

[44] *Rohan Varma*: ResNet Paper Notes

[45] *Springer Link*: A survey on Image Data Augmentation for Deep Learning

[46] *Stack Exchange*: What is batch size in neural network?

[47] *Springer Link*: Utilization of DenseNet201 for diagnosis of breast abnormality

[48] *Toward Data Science*: Review: DenseNet - Dense Convolutional Network (Image Classification)

[49] *Sergey Ioffe, Christian Szegedy, 2015*: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

[50] *Scikit-Learn*: Confusion Matrix

[51] *Kaggle*: Plot a Confusion Matrix

[52] *Scikit-Learn*: Accuracy Score

[53] *Scikit-Learn*: Classification Report

[54] *Toward Data Science*: Boost your CNN image classifier performance with progressive resizing in Keras

[55] *Cognizant*: Applying Deep Learning to Transform Breast Cancer Diagnosis