Defect Detection in Nanofibers by Image Classification

Image Analysis and Computer Vision

Politecnico di Milano

Francisco Carrillo Pérez

Contents

1 Problem Formulation

This project concerns the detection of defective regions in SEM (Scanning Electron Microscope) images. These images have been acquired for monitoring the production of nanofibers. The images are contain in the following paper [1]. Scanning Elector Microscope image with anoamlies in it An example of an image would be Figure 1. Also, we have the ground truth of the images, calculates also in [1].

So far, in [1] they have addressed the problem as an anomaly-detection problem, without exploiting during the learning (i.e. training) stage any example of defective regions. So the aim of this project is to address the defect-detection problem as a two-class classification problem where a test image is divided in patches (small squared regions) and each patch is classified as normal/anomalous.
In total there are 46 images where 40 of them contains anomalies, as in Figure 1 ,and 6 are completely normal images. So the different aims of the projects are:

- Taking patches based in the GT images where the whole patch is anomalous, or all is normal.
- Training a classifier for predicting between anomalous or normal using a Deep Learning approach.
- Using this classifier to predict each patch of a new image.

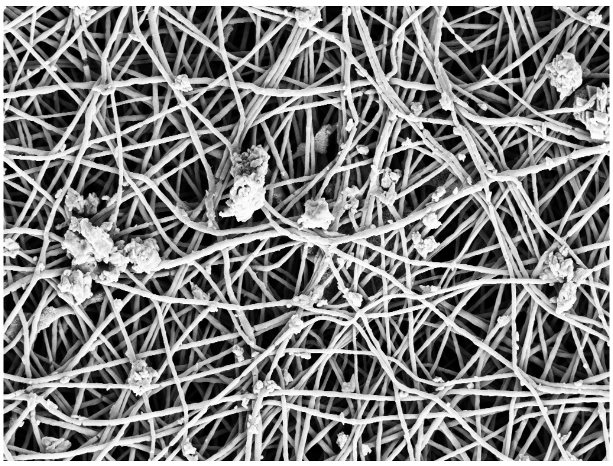


Figure 1: Example of Scanning Elector Microscope image with anoamlies in it

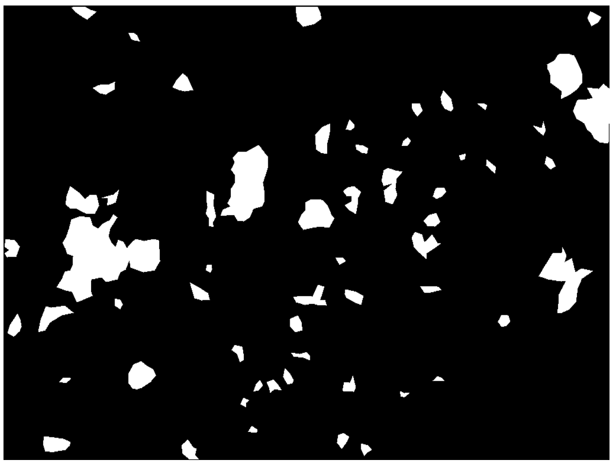


Figure 2: Example of Ground Truth image of Figure 1

2 State of the art

Noways, Image Classification is one of the hottest areas of researching. In the latest years, huge advances have been made in this field. The field where all the huge advances are being made is Deep Learning.

“Deep learning (also known as deep structured learning, hierarchical learning or deep machine learning) is a branch of machine learning based on a set of algorithms that attempt to model high level abstractions in data ”[2].

The main technique in the field of Deep Learning for image classifications are the Convolutional Neural Networks(or CNNs).

2.1 Convolutional Neural Networks

A CNN is a type of neural network for processing data that has a known, grid-type topology. They use a mathematical operation called *convolution*, therefore:

“CNN are just neural networks that use convolution in place of general matrix multiplication in at least one of their layers. ”[2]

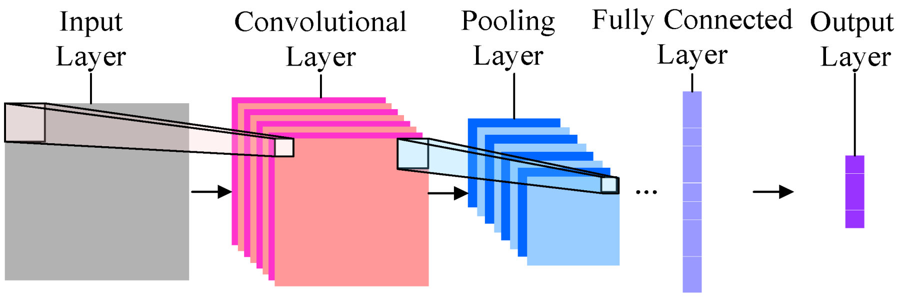


Figure 3: Example of a CNN architecture [3]

So the architecture is basicly, some layers for the convolution operation, some layers for the pooling operation and finally a fully conected layer that gives us the output.
There are some operations that are basic in a CNN:

- **Convolution**: A convolution is an integral that expresses the amount of overlap of one function g as it is shifted over another function f. It therefore ”blends” one function with another. [2]
- **Pooling**: A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs [2]. For example, one very used function, and the one that is going to be used in the code, is the max pooling operation which reports the maximun output within a rectangular neighborhood.

Also, the combination of preventing overfitting techniques (f.e. *dropout*) and CNNs give us great results in the overfitting problem.

2.2 State of the art in Convolutional Neural Networks for Image Classification

We could consider the LSVRC(Large Scale Visual Recognition Challenge) as the principal way of defining the state of the art. In this challenge, the teams have to work with the Image Net database [4], which is composed with 14,197,122 images (this number of images consoltud the February 15, 2017).

In the following chapter [5], we could see the comparation between the latest Deep Learning techniques applied to Image classification. And we could appreciate that the majority of the architectures are the combination of several ConvsNets, where there are several layers with a huge amount of neurons.

3 Solution approach

In this section I'm going to describe the different approaches that I have taken for developing this project.
Since the project aims to build a classifier using Deep Learning techniques, I decided to use a Convolutional Neural Network as the model for building the classifier, for the reasons I've described in section 2. Also, some ideas have been taking from [6].
In the following subsections I'm going to describe how the solution have been approached.

3.1 Programming language and libraries

For it's versatility and easy use I've decided to use Python. In this case, the version of Python that I'm currently using is **3.5** because is the last one supported for the library that I'm going to use.
The library I've decided to use is **Keras**, which is the High Level API for Tensorflow. At the beginning of the project, I have used Tensorflow directly, but using Keras makes everything more clear and easier.
Tensorflow is a library made by Google which works for developing Deep Learning projects. Keras is running over Tensorflow, and it was made by a Google former employee (*François Chollet*) and it had become the official High Level API for Tensorflow.
An accurate list of all the requirements of the project is the following (all of them could be installed from the *requirements.txt* file):

- **Python 3.5**
- **image==1.5.5**
- **Keras==1.2.2**
- **matplotlib==2.0.0**
- **numpy==1.12.0**
- **scipy==0.18.1**
- **tensorflow==0.12.1**
- **Pillow==4.0.0**

3.2 Generating the Data Set

First of all, I had to create a sufficiently big data set for training the network.
Since I'm taking as a reference the research made in [1], the patches that I'm going to take for training the network are going to be those where in the ground truth images 2, they are all 0 or 1, labeling them as anomalous if they are all 1, or normal if they are all 0.
For taking the patches I've decided to use a simple Matlab's script(*take_patches_from_images.m*):

```
%% CROP
clear
I = imread(path_of_the_image);
M = imread(path_of_the_gt_image);
vec = -14:14;
counter_of_normal_patches = 1;
counter_of_anomalous_patches = 1;
counter_of_patches = 16000;
for x=1:1024
        for y=1:696
                %This is the pixel were the patch is going to be crop
                    around
        pixel=I(x,y);
        pixel_gt = M(x,y);
        if(x < 29 || y < 29)

        else
            patchI = I(x+vec,y+vec);
            patch_gt = M(x+vec,y+vec);
            if(all(patch_gt))

                if(counter_of_anomalous_patches < 200)
                    path = strcat(path_for_anomalous_patches,int2str(counter_of_patches));
                    imwrite(patchI,strcat(path,'.png'));
                    counter_of_anomalous_patches = counter_of_anomalous_patches + 1;
                    counter_of_patches = counter_of_patches + 1;
                end

            else

                if(not(all(patch_gt)))

                    if(counter_of_normal_patches < 1)
                        path = strcat(path_for_normal_patches,int2str(counter_of_patches));
                        imwrite(patchI,strcat(path,'.png'));
                        counter_of_normal_patches = counter_of_normal_patches + 1;
                        counter_of_patches = counter_of_patches + 1;
                    end

                end

            end


        end
    end

end
```

With this, I've obtained 4,825 anomalous patches and 5,325 normal patches, which make a total of 10,150 patches for the data set. I have obtained the patches from 31 images. All of them are anomalous images, because from them I could take both anomalous and normal patches. The size of the patches is **29x29**.

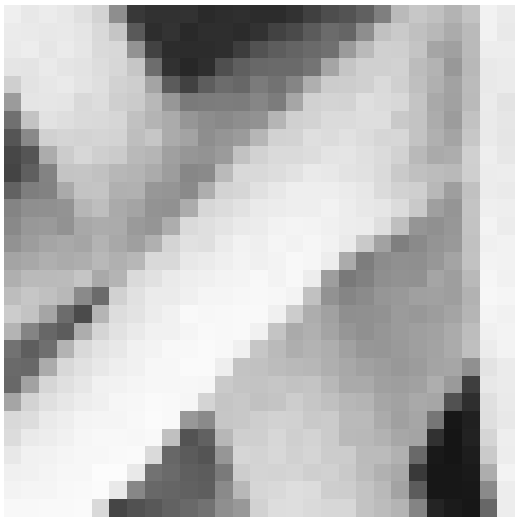


Figure 4: Example of normal patch

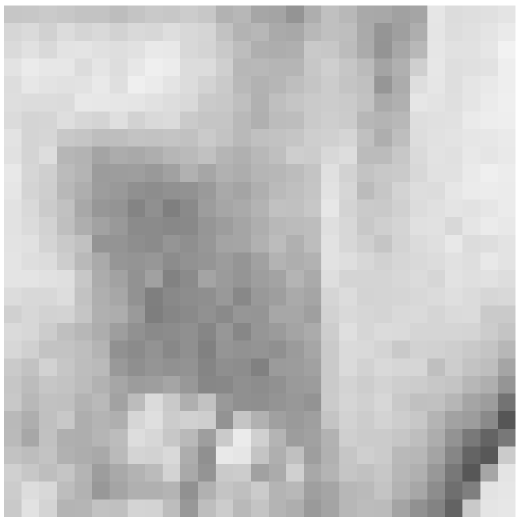


Figure 5: Example of anomalous patch

Then, I subdivided them in train, validation and test:

- **Train**: 4,807 anomalous patches and 5,307 normal patches.
- **Validation**: 9 anomalous patches and 9 normal patches.
- **Test**: 9 anomalous patches and 9 normal patches.

For the training, validation and test sets, during the execution, I have applied data augmentation as it is explained in subsection 3.5.

For testing the classifier in a whole image, I've used 9 images which contains anomalies in them 4.2.

3.3 The architeccture of the CNN

The next step was to decide how the architecture of the network should be. As it is described in section 2.1, normally a CNN is composed with various convolution and pooling layers where you are reducing the size of the image but increasing its dimensionality.
So, the architecture decided was 6:

- **First Convolutional Layer**: input with shape (29,29,3) with is the patch width, the patch height and the number of channels of the image. Then, the number of dimensions as ouput of this layer is 16 and the size of the convolution window is (5,5). The activation function is ReLU.
- **Max Pooling Layer**: max pooling layer with the pool size (2,2).
- **Second Convolutional Layer**: input with the shape of the output of the previous max pooling layer (Keras work wiht the shapes without you having to define them manually, which is quite useful). Then, the number of dimensions as ouput of this layer is 32 and the size of the convolution window is (5,5). The activation function is ReLU.
- **Max Pooling Layer**: max pooling layer with the pool size (2,2).
- **Third Convolutional Layer**: the output dimension is 64 in this case, and the convolution window has to be changed to (3,3). The activation function is also a ReLU.

- **Max Pooling Layer**: max pooling layer with the pool size (2,2).
- **Fully Connected Layer**: composed by 64 neurons, with also ReLU as activation function.
- **Dropout Layer**: droput layer to prevent overfitting with a 0.6 of keeping probability.
- **Output Layer**: output layer with the sigmoid function as activation function.

```
print("Creating the model")

print("creating first layer")
model = Sequential()
model.add(Convolution2D(dimension_first_conv, 5, 5, input_shape=(
    IMAGE_WIDTH, IMAGE_HEIGHT,3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

print("creating second layer")
model.add(Convolution2D(dimension_second_conv, 5, 5))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

print("creating third layer")
model.add(Convolution2D(dimension_fc, 3, 3))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

print("creating fc and output layer")
model.add(Flatten())
model.add(Dense(dimension_fc))
model.add(Activation('relu'))
model.add(Dropout(0.6))
model.add(Dense(1))
model.add(Activation('sigmoid'))
```

3.4 How to compute accuracy and the optimazer

The most used error function for computing the loss and the accuracy in classification problems is **cross-entropy** so this was the one selected for the training of the network.
For the optimazer I decided to use the AdamOptimazer [7], because is also one of the most used optimazer in binary classification problems.

```
model.compile(loss='binary_crossentropy',
optimizer='Adam',
metrics=['accuracy'])
```

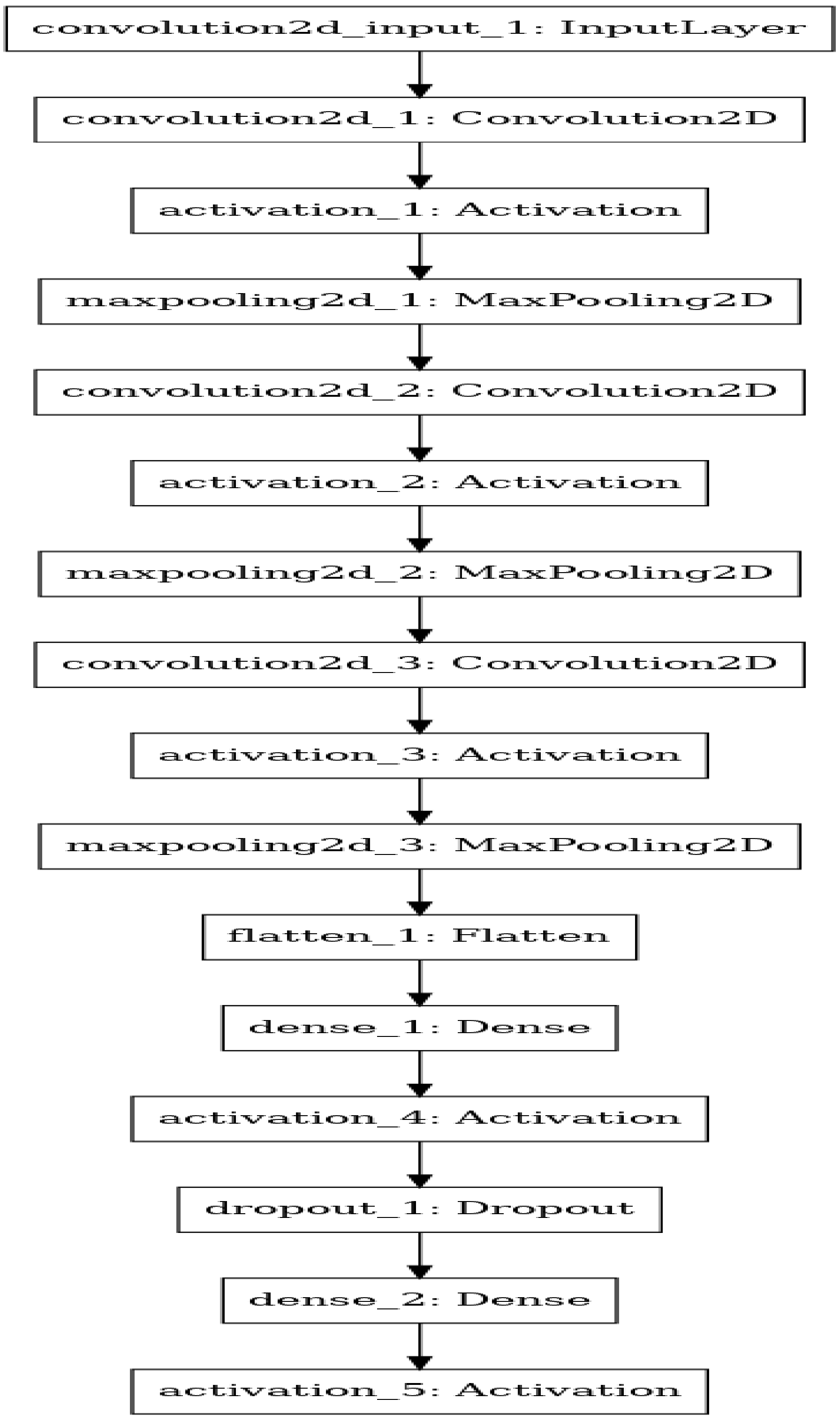

convolution2d_input_1: InputLayer
convolution2d_1: Convolution2D
activation_1: Activation
maxpooling2d_1: MaxPooling2D
convolution2d_2: Convolution2D
activation_2: Activation
maxpooling2d_2: MaxPooling2D
convolution2d_3: Convolution2D
activation_3: Activation
maxpooling2d_3: MaxPooling2D
flatten_1: Flatten
dense_1: Dense
activation_4: Activation
dropout_1: Dropout
dense_2: Dense
activation_5: Activation

Figure 6: Architecture of the CNN

3.5 Data Augmentation

Since the approach for solving the problem is using Deep Learning, we need to have a big data set, that is where Deep Learning techniques work their bests. For increasing the data set size without the necesity of taking new patches from the original images, what I decided to do was to perform data augmentation. This is a very good solution because is going to make the network more stable

to image variations and it is almost free in computation terms.
Data augmentation is very easy to do with Keras. For the training set I decided to perform scaling and an horizontal flip. For the validation adn test sets I decided to only do rescaling, to see how the network performs to data with transformations:

```
print("augmentation configuration for training")
# this is the augmentation configuration we will use for training
train_datagen = ImageDataGenerator(
                            rescale=1./255,
                            shear_range=0.2,
                            zoom_range=0.2,
                            horizontal_flip=True)

print("augmentation configuration for testing")
# this is the augmentation configuration we will use for testing:
# only rescaling
test_datagen = ImageDataGenerator(rescale=1./255)
```

3.6 Preventing overfitting: Early Stopping

Another way of preventing overfitting is using another technique which is call **Early Stopping** [8]. This technique will stop the training if the loss on the training set is going down but it is aumenting in the validation set. In *Keras* we have a function implemented for this. We could set a *patience*, which means that we will wait n epochs before stopping.

```
#Using the early stopping technique to prevent overfitting
earlyStopping= keras.callbacks.EarlyStopping(monitor='val_loss',
    patience=50, verbose=1, mode='auto')
```

3.7 Training of the network

Training the network in Keras is really easy, we have to call a function where there are some parameters that we have to adjust depending on our needs.
This will train the network using batches, which is necessary because if we don't use batches we would need to load all the training and validation set into memory, which is not a good approach since we would run out of memory space.

```
print("Fitting the model")
history = model.fit_generator(
         train_generator,
         samples_per_epoch=1000,
         callbacks=[earlyStopping],
         nb_epoch=NUM_EPOCHS,
         validation_data=validation_generator,
         nb_val_samples=nb_validation_samples)
```

After the training I decided to save the weights that the network have learned, so I don't have to trained the network everytime I want to predict new images. This is also very simple in Keras, we can save them using a single line. For loading them if restore option is selected 4.1, the file needs to has the same name:

```
print("Saving the weights")
model.save_weights('weights.h5')  # always save your weights after
    training or during training
```

3.8 Testing the accuracy

For testing the accuracy I have used the test set previously described in subsection 3.2. With a single line, we could get the loss and the accuracy on the test set:

```
test_loss = model.evaluate_generator(test_generator,val_samples =
    nb_test_samples)
print("Loss and accuracy in the test set: Loss %g, Accuracy %g"%(
    test_loss[0],test_loss[1]))
```

3.9 How to predict a whole image?

At this point, that the network is already trained and evaluated, I needed to decide how I was going to evaluate a whole image.
Since the network is trained to predict patches, we need to feed it with patches. So first, I decided to read the images as numpy arrays (I'm reading both the original image and the ground truth image):

```
data = PIL.Image.open(image_path+image_name).convert('RGB') #Reading
    the original image
data_gt = PIL.Image.open(image_path+image_name_gt).convert('RGB') #
    Reading the ground truth image
data = numpy.array(data) #Converting the image to a numpy array
data_gt = numpy.array(data_gt) #Converting the image to a numpy array
```

Since I need to plot the predictions in a new image, I created a new array with the sizes of the ground truth images (696x1024):

```
#This is the matrix for defining the image of predictions for each
    pixel.
E_image = numpy.zeros((696,1024))
```

Then with two for-nested loops, I'm going through the rows and colums of the image taking a 29x29 patch, feeding the model with it and getting the prediction for that patch. After getting the prediction I print that results in the prediction image's matrix:

```
for x1 in range(0,696,29):
  for y1 in range(0,1024,29):
        #if i'm out of boundaries don't do anything
        if(x1 + 29 > 696 or y1 + 29 > 1024):
                whylord = 0 #Just need to do something
        else:
                #otherwhise
                #Take the patch from the image
                rect = numpy.copy(data[x1:x1+IMAGE_HEIGHT,y1:y1+
                    IMAGE_HEIGHT])
                #convert it to numpy array
                rect = numpy.array(rect)
                #expand first dimension to represent batch size
                rect = numpy.expand_dims(rect,axis=0)
                #resize it
                numpy.resize(rect,(1,29,29,3))
                rect.astype(float)
                #get the prediction for the patch
                patch_prediction = model.predict(rect,verbose=0)
                #assign this prediction to the prediction image's
                    matrix
                E_image = assign_to_matrix(patch_prediction[0], E_image
                    ,x1,y1,IMAGE_HEIGHT,IMAGE_WIDTH)
```

assign_to_matrix is a method I've implemented to assign the value of the prediction to the corresponding patch in the prediction image's matrix. So it would be all 1 if the prediction is anomalous(what would be represented with white color) and all 0 if the prediction is normal (what would be represented with black color):

```
'''Assign the value of the prediction to the whole patch of an image'''
def assign_to_matrix(patch_prediction, matrix,x,y,patch_height,
    patch_width):

        '''This is because the program is taking anomalous as 0 and
            normal as 1, and when
        we want to paint it, it should be the other way around'''
        color = 0
        if(patch_prediction == 0):
          color = 1 #anomalous
        elif(patch_prediction == 1):
          color = 0 #normal

        for i in range(x,x+patch_width):
          for j in range(y,y+patch_height):
                matrix.itemset((i,j),color)

        return matrix
```

4 Experimental results

In the following subsections I'm going to talk about the results I have obtained. For doing the experiments, the Python script **IACVProject-FranciscoCarrilloPerez.py** should be in the project root folder, because is going to take the images with the ./ which means that the script is in the same folder as the folder where the images are contained. Also, the weights of the obtained results are also in the main folder, for loading them if we want. For executing it you have to do:

```
python3.5 IACVProject-FranciscoCarrilloPerez.py
```

4.1 Parameters values

The values for the dataset are already explained in subsections 3.2 and 3.5. But there's other parameters that not have been already defined:

```

"""
Different parameters that allow to change variables of the whole
    network.

@param: BATCH_SIZE_TRAIN, is going to depend on the number of samples
    that we have
@param: IMAGE_HEIGHT, height of the images
@param: IMAGE_WIDTH, width of the images
@param: IMAGE_WIDTH_ORIGINAL, original width of the SEM images
@param: IMAGE_HEIGHT_ORIGINAL, original height of the SEM images
@param: dimension_first_conv, how many dimensions do you want in the
    first convolutional layer of the network
@param: dimension_second_conv, how many dimensions do you want in the
    second convolutional layer of the network
@param: dimension_fc, how many neurons do you want in the fully
    conected layer

"""

BATCH_SIZE_TRAIN = 1000
```

```
NUM_EPOCHS = 200
IMAGE_HEIGHT =29
IMAGE_WIDTH = 29
IMAGE_WIDTH_ORIGINAL = 1024
IMAGE_HEIGHT_ORIGINAL = 696
dimension_first_conv = 16
dimension_second_conv = 32
dimension_fc = 64
```

This are the parameters I've used for getting the results that I'm going to explain in 4.2.
Also, there are this three options in the Python script to decide what we want to perform. They are boolean values that we could change depending if we want to perform training, test and restore the model:

```

'''Options for performing training, restore a model or test'''
restore = True
train = False
test = True
```

Depending on the values we are going to perform this operations or not.
I have to define the path were the data set is. Another thing that I have to explained is that the dataset must be organized in an specific way to be read by Keras. The way the data set is represented is the following:

```
- test:
      - anomalous:
           patch1.png
           patch2.png
           ...
           ...
      - normal:
           patch1.png
           patch2.png
           ...
           ...
- train:
      - anomalous:
           patch1.png
           patch2.png
           ...
           ...
      - normal:
           patch1.png
           patch2.png
           ...
           ...
- validation:
       - anomalous:
           patch1.png
           patch2.png
           ...
           ...
       - normal:
           patch1.png
           patch2.png
           ...
```

...

This way, Keras will automatically recognize each label and would assign it a number. In this case, because anomalous folder is the first one, **anomalous patches would be labeled as 0 and normal patches would be represented as 1**.
The names of the paths are respresented in the following variables:

```

'''
Full path of the different directories for loading the dataset to the
    network, and also values of the dataset.

@param: train_data_dir, full path where the training samples are
@param: validation_data_dir, full path where the validation samples are
@param: test_data_dir, full path where the validation samples are
@param: nb_train_samples, number of train samples
@param: nb_validation_samples, number of validation samples
@param: nb_test_samples, number of test samples

'''
train_data_dir = './DataSet/keras_dataset/train'
validation_data_dir = './DataSet/keras_dataset/validation'
test_data_dir = './DataSet/keras_dataset/test'
nb_train_samples = 4807 + 4664
nb_validation_samples = 9 + 9
nb_test_samples = 9 + 9
```

For changing the original image that is going to be predicted, we have to change the following lines in the script:

```

'''
Here the full images are going to be predicted

@param: image_patch, where the original iamges for testing are, with
    their gt images
@param: image_name, name of the original image
@param: image_name_gt, name of the gt image corresponding to the
    original image

'''

image_path = "./DataSet/NormalImages/Test/"
image_name = "ITIA1115.jpg"
image_name_gt = "ITIA1115_gt.png"
```

The patience for **early stopping was set to 100**.

4.2 Results of the training

With the parameters commented in the subsection 4.1, the results on the validation and in the test sets have been **100%** of accuracy in both of them.
With the firt experiment, the results where of **94,444%** but training a little bit further, the previous commented accuracy was obtained(Figures 9 and 10).

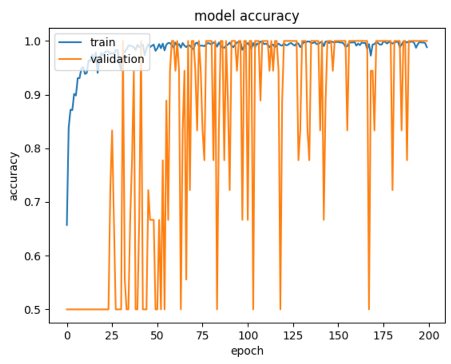


Figure 7: Accuracy on train and validation sets over 200 epochs

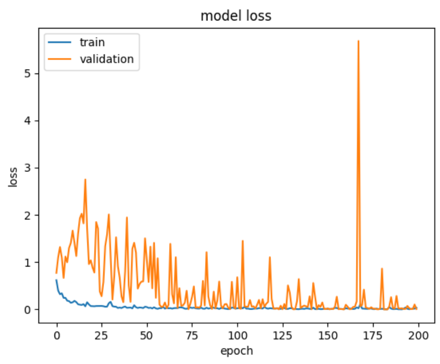
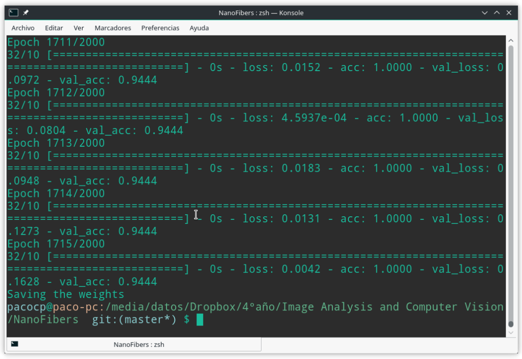


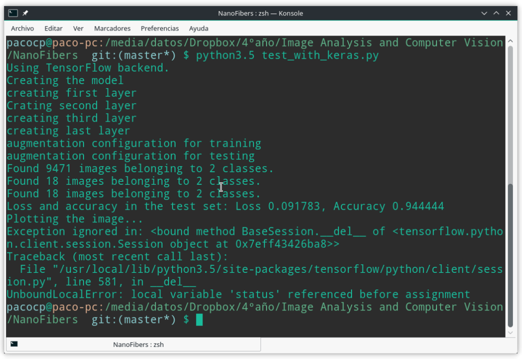
Figure 8: Loss on train and validation sets over 200 epochs

First I'm going to show the output results for the 94,44% accuracy.


NanoFibers : zsh — Konsole
Archivo Editar Ver Marcadores Preferencias Ayuda
NanoFibers : zsh

```
Epoch 1711/2000
32/10 [=====================================================================
==========================] - 0s - loss: 0.0152 - acc: 1.0000 - val_loss: 0
.0972 - val_acc: 0.9444
Epoch 1712/2000
32/10 [=====================================================================
==========================] - 0s - loss: 4.5937e-04 - acc: 1.0000 - val_los
s: 0.0804 - val_acc: 0.9444
Epoch 1713/2000
32/10 [=====================================================================
==========================] - 0s - loss: 0.0183 - acc: 1.0000 - val_loss: 0
.0948 - val_acc: 0.9444
Epoch 1714/2000
32/10 [=====================================================================
==========================] - 0s - loss: 0.0131 - acc: 1.0000 - val_loss: 0
.1273 - val_acc: 0.9444
Epoch 1715/2000
32/10 [=====================================================================
==========================] - 0s - loss: 0.0042 - acc: 1.0000 - val_loss: 0
.1628 - val_acc: 0.9444
Saving the weights
pacocp@paco-pc:/media/datos/Dropbox/4°año/Image Analysis and Computer Vision
/NanoFibers  git:(master*) $ 
```

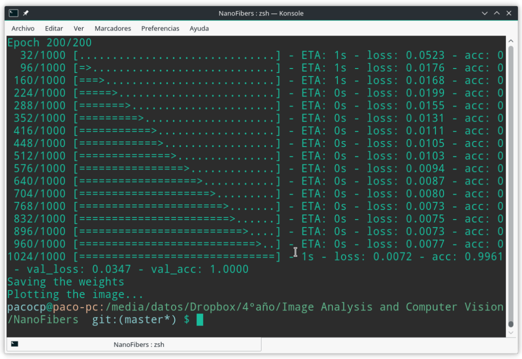
Figure 9: Output acccuracy on train and validation sets over 200 epochs. (94,444% accuracy)


NanoFibers : zsh — Konsole
Archivo Editar Ver Marcadores Preferencias Ayuda
NanoFibers : zsh

```
pacocp@paco-pc:/media/datos/Dropbox/4°año/Image Analysis and Computer Vision
/NanoFibers  git:(master*) $ python3.5 test_with_keras.py
Using TensorFlow backend.
Creating the model
creating first layer
Crating second layer
creating third layer
creating last layer
augmentation configuration for training
augmentation configuration for testing
Found 9471 images belonging to 2 classes.
Found 18 images belonging to 2 classes.
Found 18 images belonging to 2 classes.
Loss and accuracy in the test set: Loss 0.091783, Accuracy 0.944444
Plotting the image...
Exception ignored in: <bound method BaseSession.__del__ of <tensorflow.pytho
n.client.session.Session object at 0x7eff43426ba8>>
Traceback (most recent call last):
  File "/usr/local/lib/python3.5/site-packages/tensorflow/python/client/sess
ion.py", line 581, in __del__
UnboundLocalError: local variable 'status' referenced before assignment
pacocp@paco-pc:/media/datos/Dropbox/4°año/Image Analysis and Computer Vision
/NanoFibers  git:(master*) $ 
```

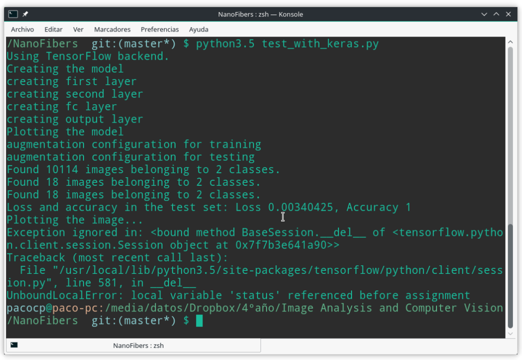
Figure 10: Output of test set with previous weights. Fig 9

And of course, the same outputs but with the weights that obtained 100% accuracy:


NanoFibers : zsh — Konsole
Archivo Editar Ver Marcadores Preferencias Ayuda
NanoFibers : zsh

```
Epoch 200/200
  32/1000 [..............................] - ETA: 1s - loss: 0.0523 - acc: 0
  96/1000 [=>............................] - ETA: 1s - loss: 0.0176 - acc: 0
 160/1000 [===>..........................] - ETA: 1s - loss: 0.0168 - acc: 0
 224/1000 [=====>........................] - ETA: 0s - loss: 0.0199 - acc: 0
 288/1000 [=======>......................] - ETA: 0s - loss: 0.0155 - acc: 0
 352/1000 [=========>....................] - ETA: 0s - loss: 0.0131 - acc: 0
 416/1000 [===========>..................] - ETA: 0s - loss: 0.0111 - acc: 0
 448/1000 [============>.................] - ETA: 0s - loss: 0.0105 - acc: 0
 512/1000 [==============>...............] - ETA: 0s - loss: 0.0103 - acc: 0
 576/1000 [================>.............] - ETA: 0s - loss: 0.0094 - acc: 0
 640/1000 [==================>...........] - ETA: 0s - loss: 0.0087 - acc: 0
 704/1000 [====================>.........] - ETA: 0s - loss: 0.0080 - acc: 0
 768/1000 [======================>.......] - ETA: 0s - loss: 0.0073 - acc: 0
 832/1000 [=======================>......] - ETA: 0s - loss: 0.0075 - acc: 0
 896/1000 [=========================>....] - ETA: 0s - loss: 0.0073 - acc: 0
 960/1000 [===========================>..] - ETA: 0s - loss: 0.0077 - acc: 0
1024/1000 [==============================] - 1s - loss: 0.0072 - acc: 0.9961
 - val_loss: 0.0347 - val_acc: 1.0000
Saving the weights
Plotting the image...
pacocp@paco-pc:/media/datos/Dropbox/4°año/Image Analysis and Computer Vision
/NanoFibers  git:(master*) $ 
```

Figure 11: Output acccuracy on train and validation sets over 200 epochs. (100% accuracy)


NanoFibers : zsh — Konsole
Archivo Editar Ver Marcadores Preferencias Ayuda
NanoFibers : zsh

```
/NanoFibers  git:(master*) $ python3.5 test_with_keras.py
Using TensorFlow backend.
Creating the model
creating first layer
creating second layer
creating fc layer
creating output layer
Plotting the model
augmentation configuration for training
augmentation configuration for testing
Found 10114 images belonging to 2 classes.
Found 18 images belonging to 2 classes.
Found 18 images belonging to 2 classes.
Loss and accuracy in the test set: Loss 0.00340425, Accuracy 1
Plotting the image...
Exception ignored in: <bound method BaseSession.__del__ of <tensorflow.pytho
n.client.session.Session object at 0x7f7b3e641a90>>
Traceback (most recent call last):
  File "/usr/local/lib/python3.5/site-packages/tensorflow/python/client/sess
ion.py", line 581, in __del__
UnboundLocalError: local variable 'status' referenced before assignment
pacocp@paco-pc:/media/datos/Dropbox/4°año/Image Analysis and Computer Vision
/NanoFibers  git:(master*) $ 
```

Figure 12: Output of test set with previous weights. Fig 11

4.3 Filters the convolutional layers are learning

Here I'm going to show the most significant filters that the second and the third convolutional layers are learning. I hadn't been able to plot the filters for all the layers. They have been plot with a Python script which could also been found in the main folder(*obtain_filters.py*).

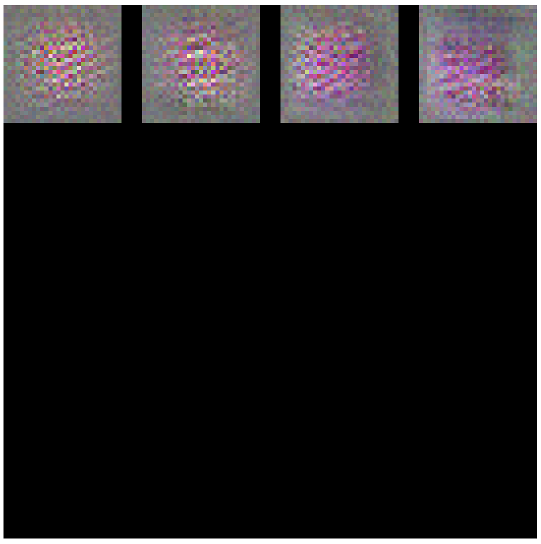


Figure 13: Filters that the second convolutional layer have learned

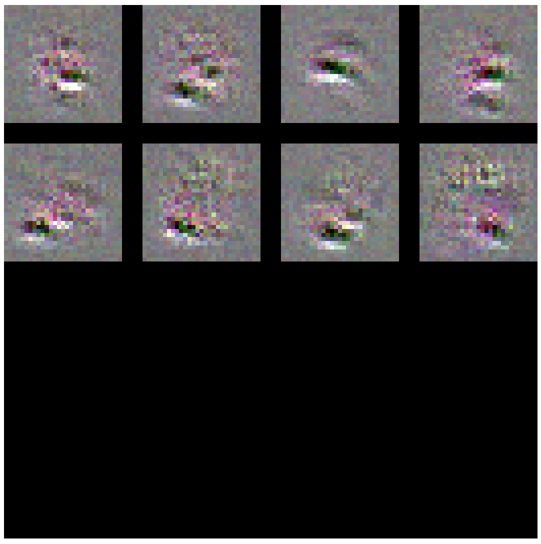


Figure 14: Filters that the third convolutional layer have learned

4.4 Results evaluating original images

Here I'm going to show the results obtained in the following test images which are contained in the NormalImages/Test folder. All the images could be seen in full resolution at the Images/Predictions/ folder, because in the document they are flatten because of LaTeXproperties.

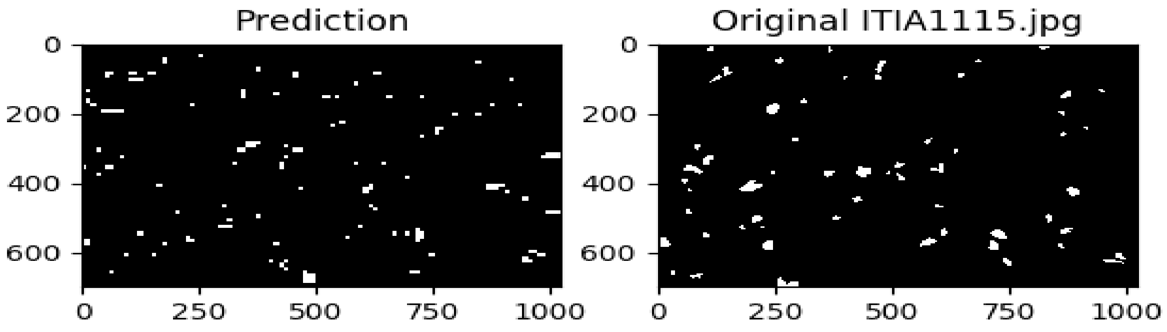


Figure 15: Prediction in the ITIA1115 image.

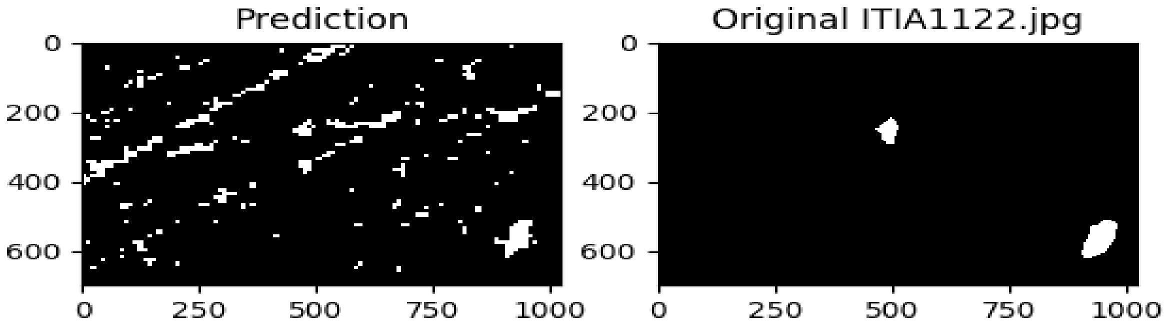


Figure 16: Prediction in the ITIA1122 image.

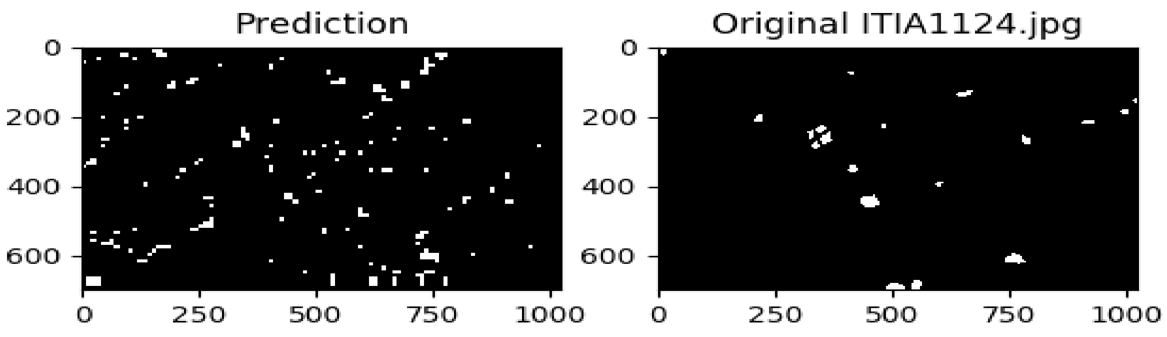


Figure 17: Prediction in the ITIA1124 image.

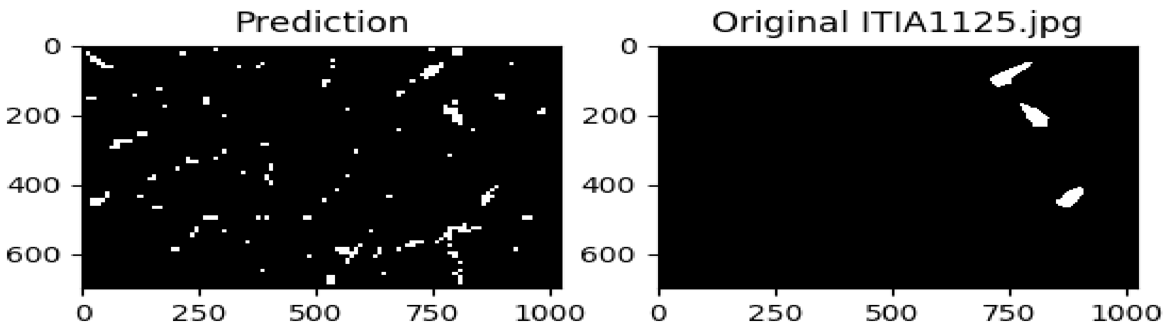


Figure 18: Prediction in the ITIA1125 image.

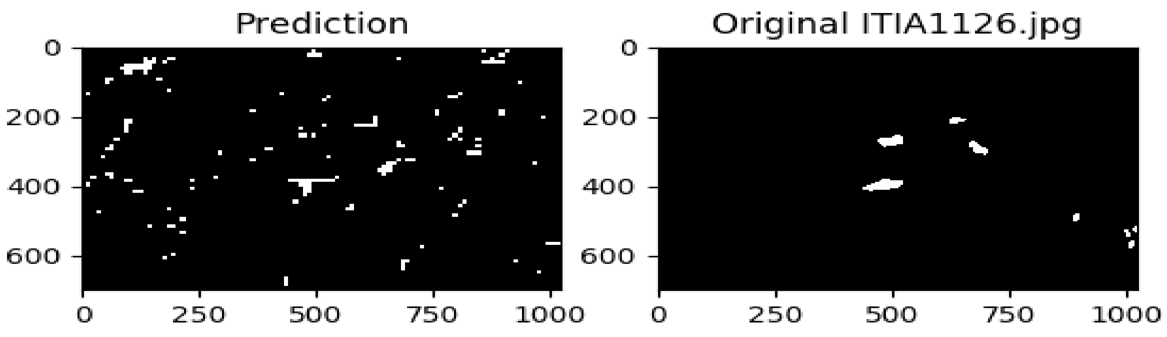


Figure 19: Prediction in the ITIA1126 image.

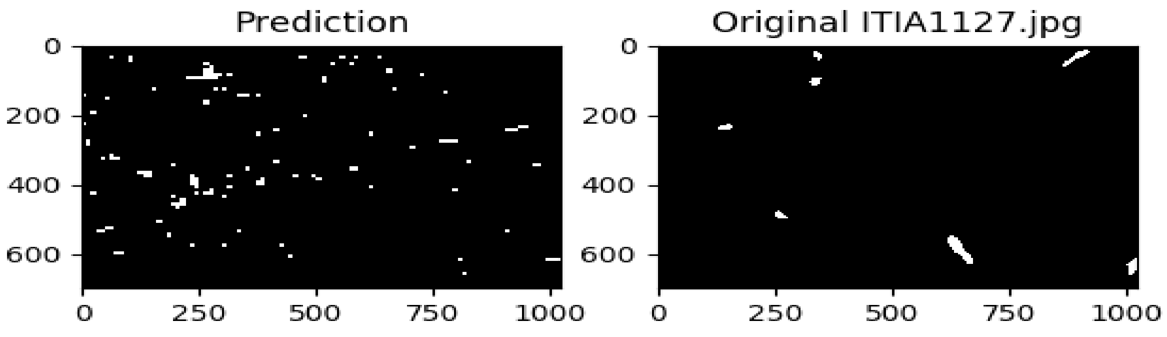


Figure 20: Prediction in the ITIA1127 image.

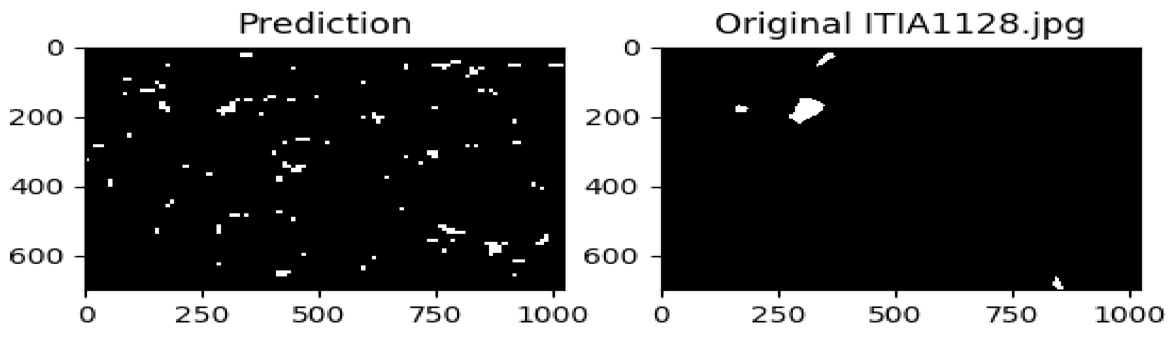


Figure 21: Prediction in the ITIA1128 image.

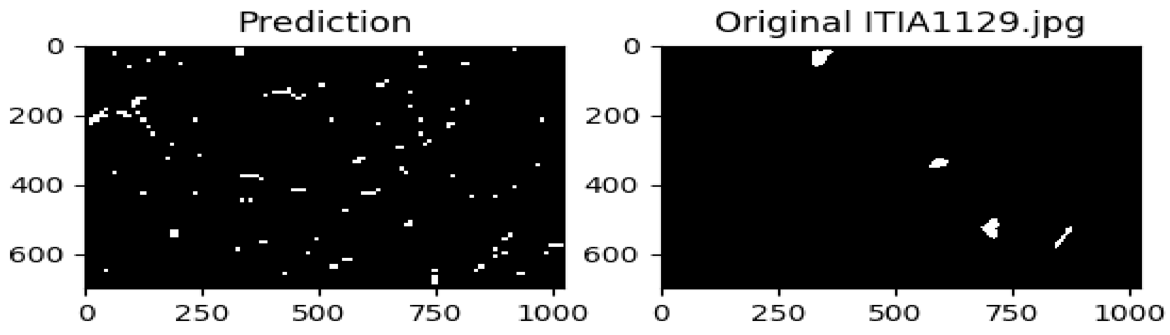


Figure 22: Prediction in the ITIA1129 image.

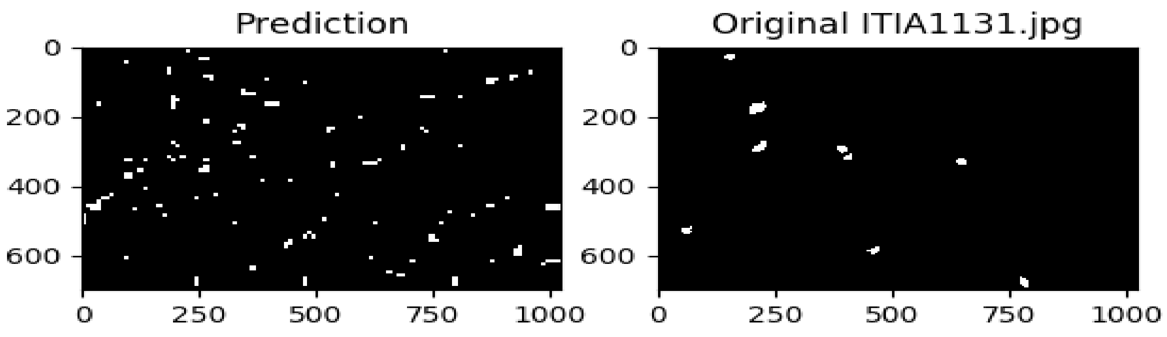


Figure 23: Prediction in the ITIA1131 image.

4.5 Compare accuracy

For lack of time, I have not been able to perform the comparation between the prediction image and the whole image. But the algorithm for compare them would be:

```
%Some rules would be included for no taking values out of the matrixes

count_correct_normal = 0
count_correct_anomalous = 0
count_false_positive = 0
count_false_negative = 0

for i from 0 to 696
    for j from 0 to 1024
        patch_gt = take 29x29 patch around pixel(i,j) in ground truth from the original image
        patch_prediction = take 29x29 patch around pixel(i,j) in the prediction image
        if(patch_gt is all anomalous)
           if(patch_prediction is all anomalous)
               count_correct_anomalous += 1
           elif(patch_prediction is all normal)
               count_false_positive += 1
        elif(patch_gt is all normal)
            if(patch_prediction is all anomalous)
                 count_false_negative += 1
            elif(patch_prediction is all normal)
                 count_correct_normal += 1
```

5 Conclusions

As we see in the previous sections the classifier works very well classifying single patches, taking as reference the accuracy and loss in the validation and test sets, but it didn't perform it's best predicting a whole image.
This could happend for various reasons:

- The number of samples in the dataset is not enough to generalize. Since we are working with 10,114 patches, it should not be a problem, but maybe the diversity of them is not enough.
- Since I'm only considering patches where the whole patch is anomalous or normal, when the network is predicting the whole image,there is a high probability that, if it has to classify a patch which has anomalies and normal parts in it, is not going to predict it correctly, just because it hasn't seen anything like that before.

Both of them could approach with the improvement of the dataset.
Eventhough, the network has more false positives that false negatives, what is better that the other way around (None evaluation have been performed for lack of time, but the algorithm could be found in subsection 4.5).
Eventhough the final results, we could see that with a relatively small dataset,a small network and not much time for training we have been able to get a **100%** accuracy in both validation and test sets, which are impresive results. Improving the dataset I think that in the future this could be use for classifying a whole image with the same impresive results as the one obtained in the training phase.

References

[1] D. Carrera, F. Manganini, G. Boracchi, and E. Lanzarone, “Defect detection in sem images of nanofibrous materials,” *IEEE Transactions on Industrial Informatics*.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[3] “Cnn architecture.” `http://www.mdpi.com/information/information-07-00061/article_deploy/html/images/information-07-00061-g001.png`. Accessed: 2017-02-12.

[4] “Imagenet database.” `http://www.image-net.org/`. Accessed: 2017-02-12.

[5] R. T. Ionescu and M. Popescu, *State-of-the-Art Approaches for Image Classification*, pp. 41–52. Cham: Springer International Publishing, 2016.

[6] D. C. Cires¸an, L. M. Gambardella, A. Giusti, and J. Schmidhuber, “Deep neural networks segment neuronal membranes in electron microscopy images,” in *In NIPS*, pp. 2852–2860, 2012.

[7] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *CoRR*, vol. abs/1412.6980, 2014.

[8] S. Raudys and T. Cibas, “Regularization by early stopping in single layer perceptron training,” in *Proceedings of the 1996 International Conference on Artificial Neural Networks*, ICANN 96, (London, UK, UK), pp. 77–82, Springer-Verlag, 1996.