The Eye Fundus Image Analysis

Using Deep Convolutional Neural Netoworks

Mohammad Huzefa Shaikh
M.Sc. Data Science and Computational Intelligence
8012085

Abstract— This paper explicates a study on the eye fundus Images, a retinal image analysis; which, as a result, provides segmentation of the retinal blood vessels. The Deep Convolutional Neural Networks were used to perform the extraction and analysis; which, in recent years, has been recognized as one of the most successful Artificial Neural Networks' methods, specifically for the feature extraction and the analysis of images. The CNN model was trained with the patches of the eye fundus images and the blood vessel ground truths (examples made by human-experts) of the same images. Various datasets of the eye fundus images are publically available, nonetheless, the DRIVE dataset was used in this experiment since it was easier to obtain.

INTRODUCTION

Eye Fundus Image Analysis (EFIA) is the crux of diagnosis of the eye diseases such as diabetic retinopathy, macular degeneration, and glaucoma. Specifically, the diabetic retinopathy induced by diabetes can cause serious vision impairment (blindness). The retinal (i.e. light-sensitive tissue) blood vessels are mutated (affected) by the diabetic retinopathy, which results in a complete blindness [3].

In order to identify the diabetic retinopathy, the contusion (lesion) correlated with vascular deformities due to diabetes, needs to be identified in the blood vessels [3]. The current method requires trained ophthalmologist to assess eye fundus images to extract mutations in the blood vessels, such illustrations are time-consuming and sometimes inconsistent [3]. Until now, optometrists predominantly used such a method. These human assessments were serviceable for a few diagnoses, however, they failed to deliver in many situations [2].

EFIA can be used to automatically predict the later mutations in the retinal blood vessels; such a prediction might help optometrists to conduct the forthcoming eye conditions. Deep Convolutional Neural Networks (CNNs) can enhance such diagnoses [1]. The advantage of having a 3-dimensional neuron architecture can provide deep segmentation and individual patch analysis [11]. ConvNets has produced medical diagnoses, in many cases, better than manual interpretations of the human experts [1].

LITERATURE REVIEW

In the recent years, Deep Convolutional Neural Networks have been successful in the field of Computer Vision [3, 1, 2]. Especially, the biomedical image segmentation has become a popular CNNs interpretation [1, 3, 2]. In [1] and [3], the challenges faced by the current diagnostic measurements of the diabetic retinopathy and how the CNN can enhance it, is indicated. CNNs' unique network architecture enables it to make automation of the feature design efficient, the non-linearity transformation of CNN layers' outputs makes the network effective [3].

The CNNs standard requirement is mostly for classification of the whole image (one image belonging to one class), but the current problem requires more than that, every pixel of the image needs to be classified (localization) [2]. [1] and [2] has discussed that the CNNs' initial approach to EFIA, which was essentially patch-based, did deliver to some extent but it was not adequate. It is implemented by assigning a patch around

each pixel for localization. Nevertheless, it had a few disadvantages; network slowness, due to the individual execution of each patch (local region), redundancy, caused by the extending patches, and either localization accuracy or the context usage could be achieved [2].

The fully connected Convolutional Neural Networks architecture, used by [1] and [2], is established to be efficient in EFIA. The architecture converges a base network for the process of blood vessel segmentation [1]. The network is transformed in such a way that it can get the work done only with the small amount of data (a few images) [2], a similar approach is employed for EFIA. The principle approach in [2] is to extend a typical shrinking network by progressive CNN layers and upsampling variables, containing numerous feature channels, are used instead of pooling controllers, which increases output resolution. In [1], Initially, Visual Geometry Group (VGG) network is employed, which is mostly used for large-scale images' classification. The final frames of the network, containing fully connected layers, are removed, leaving the network merely with the CNNs implemented with Rectified Linear Unit (ReLU) activations and four max-pooling layers [1]. The use of max-pooling layers distinguishing multiple serialized frames, containing various Convolutional layers [1]. The class-balancing cross entropy loss function is applied for training the network [1, 2]. However, in [3], rendering of the Regression Activation Maps (RAM) is used, RAM is formed for an input image which confines the distinct regions upon the regression inferences.

Dataset Description And Features Selection

A publically available eye fundus images' dataset, DRIVE (Digital Retinal Images for Vessel Extraction), is used for Eye Fundus Image Analysis (EFIA) [5]. The dataset's photographs were procured from a screening program of retinopathy in the Netherlands [5]. The diabetic subjects on whom the experiments were conducted, were between twenty-five to ninety years [5]. Around forty images were randomly accumulated, barely seven of them showed implications of scanty diabetic retinopathy, and thirty-three of them were declared normal [5].

Figure 1: Eye fundus image [5]

The non-mydriatic Canon CR5 3CCD camera, with a 45-degree view field (FOV), was used to obtain the images [5]. 8 bits per color plane at pixels of 768 by 584 was used as the configuration while capturing the images [5]. All the images are spherical, with the 540 pixels' diameter [5]. The segments of the images outside the field of view have been excluded, for the dataset [5]. The masks (spherical area of the images) are included in the dataset to outline the field of view [5].

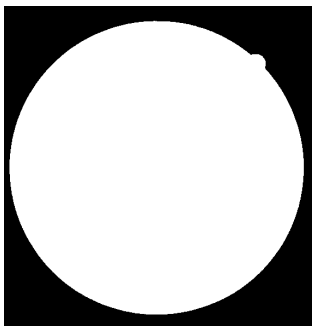

Figure 2: Image mask [5]

The dataset is already divided by [5], into two segments, for the training and testing [5]. Each part consists of 20 images, 20 manual illustrations of the blood vessels (vasculature) in the eye fundus, and 20 masks of each image [5]. The illustrated vasculatures in the testing set were used to compare with the system-generated segmentation of vasculatures [5]. An experienced ophthalmologist trained the human observers who manually performed the segmentation of vasculatures [5]. Many clinical diagnoses were actually conducted using images of the DRIVE dataset [5].

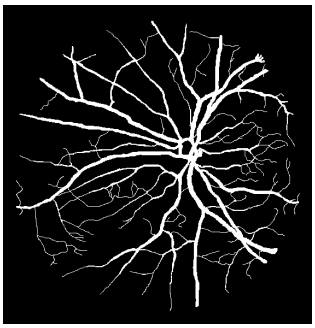

FIGURE 3: HUMAN-MADE SEGMENTATION [5]

The current experiment consists of the feature extraction (patches' acquirement) of the eye fundus images, classification of the acquired patches, and then automatic segmentation of the retinal blood vessels; analyzed on the patches. The purpose of the experiment is to observe If the computer-generated illustrations of the retinal blood vessels are as accurate as the human-made manual segmentation of (vasculature) blood vessels.

USED METHODS

Convolutional Neural Networks was the appropriate choice for the Eye Fundus Image Analysis (EFIA) since it has been proved that CNN architecture delivers the best results when it comes to image analysis and classification.

ConvNets' network operates exactly as the ANNs'. It is also feed-forward, the information flow is unidirectional, the network takes in the data as the input, process it through the network, and to the output [11]. As the ANNs are inspired by the biological Neural Networks, CNNs also emulates the brain's visual cortex, it comprises variant layers of fluctuating (simple and complex) cells [11].

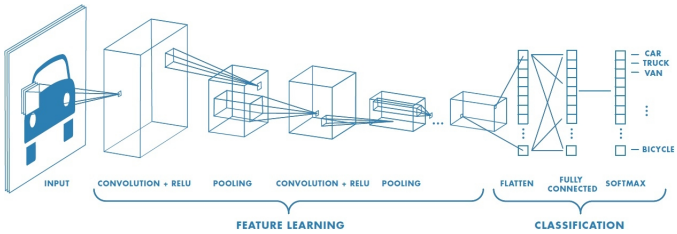

FIGURE 4: CONVNETS ARCHITECTURE [12]

The architecture is considered a feature extractor; thus it learns the representation of the image (input) features [11]. CNNs architecture reduces the computation as it's feature map's each neuron consist of a receptive field, unlike ANNs, it only connects to the neighboring neurons (a certain group of neurons) of the previous layer through the set of trainable weights, instead of every neuron [11]. Every feature map within the network carries different weights, however, each neuron within every feature map have exactly same weights [11].

- $Yk = f(Wk*x)$

Yk: feature map of the *kth* output

Wk: Convolutional filter of the *kth* feature map (pooling and extraction)

x: Input image

f: Non-linear activation function (Rectified Linear Unit in our case)

EXPERIMENTAL SETUP

PRE-PROCESSING

The DRIVE dataset's (training) images were pre-processed using the OpenCV methods, prior to the training. OpenCV is an open computer vision library, it comprises useful Computer Vision algorithms. Below are the applied transformations. *(FYI, DRIVE dataset was converted to HDF5 for making it easily accessible, go to Appendix E)*

- *adjustment of Gamma*
- *Standardization*
- *conversion to gray-scale*
- *Contrast-limited adaptive histogram equalization (CLAHE)*

Please observe figure 5 for the histogram equalization of the images and figure 6 to observe the transformed image. (go to Appendix A for the whole code)

```
#==== histogram equalization
def histo_equalized(imgs):
    assert (len(imgs.shape)==4)  #4D arrays
    assert (imgs.shape[1]==1)  #check the channel is 1
    imgs_equalized = np.empty(imgs.shape)
    for i in range(imgs.shape[0]):
        imgs_equalized[i,0] = cv2.equalizeHist(np.array(imgs[i,0], dtype = np.uint8))
    return imgs_equalized
```

FIGURE 5: HISTOGRAM EQUALIZATION

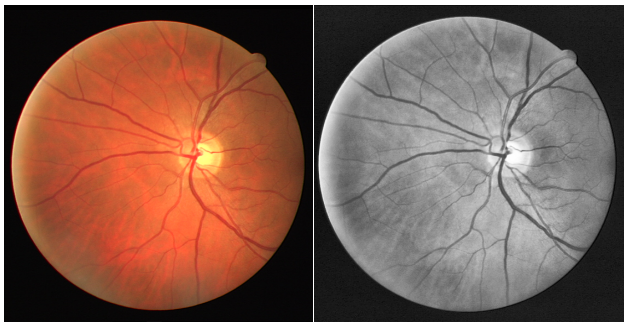

FIGURE 6: TRANSFORMED IMAGE

FEATURE EXTRACTION

The training was conducted using the extracted patches of images. To show the model the difference between blood vessels and the Field OF View (FOV), patches outside the FOV were selected. Please see *figure 7* for the python code and *figure 8* for observing a sample of extracted patches (image and its segmentation). (go to *Appendix B* for the whole code)

```
#extract patches randomly from the full training images
#  -- Inside the full image
def extract_random(full_imgs,full_masks, patch_h,patch_w, N_patches, inside=True):
    if (N_patches%full_imgs.shape[0] != 0):
        print("N_patches: plase enter a multiple of 20")
        exit()
    assert (len(full_imgs.shape)==4 and len(full_masks.shape)==4)  #4D arrays
    assert (full_imgs.shape[1]==1 or full_imgs.shape[1]==3)  #check the channel is 1 or 3
    assert (full_masks.shape[1]==1)  #masks only black and white
    assert (full_imgs.shape[2] == full_masks.shape[2] and full_imgs.shape[3] == full_masks.shape[3])
    patches = np.empty((N_patches,full_imgs.shape[1],patch_h,patch_w))
    patches_masks = np.empty((N_patches,full_masks.shape[1],patch_h,patch_w))
    img_h = full_imgs.shape[2]  #height of the full image
    img_w = full_imgs.shape[3] #width of the full image
    # (0,0) in the center of the image
    patch_per_img = int(N_patches/full_imgs.shape[0])  #N_patches equally divided in the full images
    print("patches per full image: " +str(patch_per_img))
    iter_tot = 0   #iter over the total numbe rof patches (N_patches)
    for i in range(full_imgs.shape[0]):  #loop over the full images
        k=0
        while k <patch_per_img:
            x_center = random.randint(0+int(patch_w/2),img_w-int(patch_w/2))
            # print "x_center " +str(x_center)
            y_center = random.randint(0+int(patch_h/2),img_h-int(patch_h/2))
            # print "y_center " +str(y_center)
            #check whether the patch is fully contained in the FOV
            if inside==True:
                if is_patch_inside_FOV(x_center,y_center,img_w,img_h,patch_h)==False:
                    continue
            patch = full_imgs[i,:,y_center-int(patch_h/2):y_center+int(patch_h/2),x_center-int(patch_w/2):x_center+int(patch_w/2)]
            patch_mask = full_masks[i,:,y_center-int(patch_h/2):y_center+int(patch_h/2),x_center-int(patch_w/2):x_center+int(patch_w/2)]
            patches[iter_tot]=patch
            patches_masks[iter_tot]=patch_mask
            iter_tot +=1   #total
            k+=1  #per full_img
    return patches, patches_masks
```

FIGURE 7: EXTRACTING PATCHES

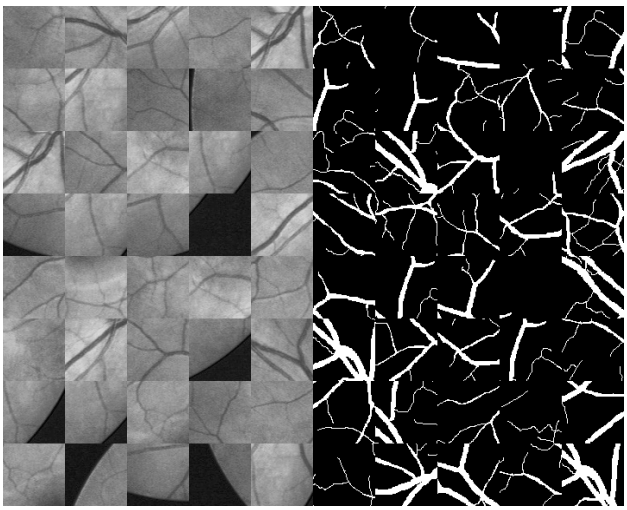

FIGURE 8: IMG AND SEGMENTATION PATCHES

TRAINING

A set of 3200 patches were considered to be extracted, for our first and second experiment at least, the results of both of them were a disaster, nevertheless, 160 patches of each training images. 90% of the set was used for the training and the rest of it was used for validation.

The cross-entropy was used as a loss function (please see *figure 9*) and the optimizer, which was used in the experiment, is the stochastic gradient descent [1]. Rectifier Linear Unit was employed, as the activation function, after every convolutional layer. 0.2 dropout between the layers was used and the training was conducted for 100 epochs, for our first experiment at least, with the batch (size) of 32 patches.

$$\mathcal{L}(\mathbf{W}) = -\beta \sum_{j \in Y_+} \log P(y_j = 1|X;\mathbf{W}) - (1-\beta) \sum_{j \in Y_-} \log P(y_j = 0|X;\mathbf{W})$$

FIGURE 9 CROSS-ENTROPY (REMOVED SUBSCRIPT *n* TO SIMPLIFY) [1]

W: *Set of CNN parameters*

β: Handler of the imbalance between background and foreground pixels' numbers

Y+ & Y-: Ground truth's (Y) foreground and background sets

- $\beta = |Y-|/|Y|$ *(for the EFIA)*

Please go through *figure 10* to observe convolutional neural networks' implementation with *Keras* and go to *Appendix C* to see the whole training code *(Keras is an open source API for Neural Networks and it is developed in python)*.

The training in the first experiment exerted around 15 hours to complete, even though it did not produce any usable results (please go to the results section for more details).

```
#Define the neural network
def get_unet(n_ch,patch_height,patch_width):
    inputs = Input(shape=(n_ch,patch_height,patch_width))
    conv1 = Conv2D(32, (3, 3), activation='relu', padding='same',data_format='channels_first')(inputs)
    conv1 = Dropout(0.2)(conv1)
    conv1 = Conv2D(32, (3, 3), activation='relu', padding='same',data_format='channels_first')(conv1)
    pool1 = MaxPooling2D((2, 2))(conv1)
    #
    conv2 = Conv2D(64, (3, 3), activation='relu', padding='same',data_format='channels_first')(pool1)
    conv2 = Dropout(0.2)(conv2)
    conv2 = Conv2D(64, (3, 3), activation='relu', padding='same',data_format='channels_first')(conv2)
    pool2 = MaxPooling2D((2, 2))(conv2)
    #
    conv3 = Conv2D(128, (3, 3), activation='relu', padding='same',data_format='channels_first')(pool2)
    conv3 = Dropout(0.2)(conv3)
    conv3 = Conv2D(128, (3, 3), activation='relu', padding='same',data_format='channels_first')(conv3)

    up1 = UpSampling2D(size=(2, 2))(conv3)
    up1 = concatenate([conv2,up1],axis=1)
    conv4 = Conv2D(64, (3, 3), activation='relu', padding='same',data_format='channels_first')(up1)
    conv4 = Dropout(0.2)(conv4)
    conv4 = Conv2D(64, (3, 3), activation='relu', padding='same',data_format='channels_first')(conv4)
    #
    up2 = UpSampling2D(size=(2, 2))(conv4)
    up2 = concatenate([conv1,up2], axis=1)
    conv5 = Conv2D(32, (3, 3), activation='relu', padding='same',data_format='channels_first')(up2)
    conv5 = Dropout(0.2)(conv5)
    conv5 = Conv2D(32, (3, 3), activation='relu', padding='same',data_format='channels_first')(conv5)
    #
    conv6 = Conv2D(2, (1, 1), activation='relu',padding='same',data_format='channels_first')(conv5)
    conv6 = core.Reshape((2,patch_height*patch_width))(conv6)
    conv6 = core.Permute((2,1))(conv6)
    ############
    conv7 = core.Activation('softmax')(conv6)

    model = Model(input=inputs, output=conv7)

    # sgd = SGD(lr=0.01, decay=1e-6, momentum=0.3, nesterov=False)
    model.compile(optimizer='sgd', loss='categorical_crossentropy',metrics=['accuracy'])

    return model
```

FIGURE 10: CONVOLUTIONAL NEURAL NETWORKS IMPLEMENTATION

TESTING

The testing was conducted on the provided 20 test images in the DRIVE dataset. Used the provided manual blood vessels' segmentation to compare with the machine-generated segmentation. (please see *figure 11* for the testing implementation and go to *Appendix D* to see what else is happening in the testing file)

```
#================ Run the prediction of the patches ==================================
best_last = config.get('testing settings', 'best_last')
#Load the saved model
model = model_from_json(open(path_experiment+name_experiment +'_architecture.json').read())
model.load_weights(path_experiment+name_experiment + '_'+best_last+'_weights.h5')
#Calculate the predictions
predictions = model.predict(patches_imgs_test, batch_size=32, verbose=2)
print("predicted images size :")
print(predictions.shape)

#===== Convert the prediction arrays in corresponding images
pred_patches = pred_to_imgs(predictions, patch_height, patch_width, "original")
```

FIGURE 11:TESTING IMPLEMENTATION

In order to identify the field of view, just so the model would only consider the pixels within it, used the provided image masks. In order to advance the performance, averaged the predictions of each pixel to procure the probability of it being a blood vessel. By using the stride of certain pixels, in width and height, extracted serialized overlapping patches from all the test images.

Onwards, by averaging probabilities through every predicted patch incorporating the pixel, acquired the blood vessel probability for all the pixels.

RESULTS

FIRST EXPERIMENT

Experiment 1 ran for almost 15 hours, even though the patches' size (N_subimgs) was reduced to 3200 (160 patches from each image), but it did not produce any results, however, it did create a nohup log file and a file containing the successful weights (visit https://github.com/Huzmorgoth/Aritificial-Neural-Networks/blob/master/ANNCourseWorkExperimentResult/ANNCourseWorkExperimentResult_training.nohup for the produced results of experiment 1). Below mentioned image shows the configuration for the first experiment.

```
[data paths]
path_local =  ./DRIVE_datasets_training_testing/
train_imgs_original = DRIVE_dataset_imgs_train.hdf5
train_groundTruth = DRIVE_dataset_groundTruth_train.hdf5
train_border_masks = DRIVE_dataset_borderMasks_train.hdf5
test_imgs_original = DRIVE_dataset_imgs_test.hdf5
test_groundTruth = DRIVE_dataset_groundTruth_test.hdf5
test_border_masks = DRIVE_dataset_borderMasks_test.hdf5


[experiment name]
name = ANNCourseWorkExperimentResult


[data attributes]
#Dimensions of the patches extracted from the full images
patch_height = 48
patch_width = 48


[training settings]
#number of total patches:
N_subimgs = 3200
#if patches are extracted only inside the field of view:
inside_FOV = False
#Number of training epochs
N_epochs = 100
batch_size = 32
#if running with nohup
nohup = True


[testing settings]
#Choose the model to test: best==epoch with min loss, last==last epoch
best_last = best
#number of full images for the test (max 20)
full_images_to_test = 20
#How many original-groundTruth-prediction images are visualized in each image
N_group_visual = 1
#Compute average in the prediction, improve results but require more patches to be predicted
average_mode = True
#Only if average_mode==True. Stride for patch extraction, lower value require more patches to be predicted
stride_height = 46
stride_width = 46
#if running with nohup
nohup = True
```

FIGURE 12: STRUCTURE FILE 1

SECOND EXPERIMENT

The second experiment ran for 7 hours. The epochs were set to 20, the images (of training and testing) were reduced to 50% of the original set, and the stride_height and stride_width were set to 46 (while testing) to reduce the running time of the training and testing.

Please observe the results mentioned below. As you can see, the results are produced but they are quite bad.

```
[data paths]
path_local =  ./NEWDRIVE_datasets_training_testing/
train_imgs_original = NEWDRIVE_dataset_imgs_train.hdf5
train_groundTruth = NEWDRIVE_dataset_groundTruth_train.hdf5
train_border_masks = NEWDRIVE_dataset_borderMasks_train.hdf5
test_imgs_original = NEWDRIVE_dataset_imgs_test.hdf5
test_groundTruth = NEWDRIVE_dataset_groundTruth_test.hdf5
test_border_masks = NEWDRIVE_dataset_borderMasks_test.hdf5


[experiment name]
name = ANNCourseWorkExperimentResult


[data attributes]
#Dimensions of the patches extracted from the full images
patch_height = 48
patch_width = 48


[training settings]
#number of total patches:
N_subimgs = 3200
#if patches are extracted only inside the field of view:
inside_FOV = False
#Number of training epochs
N_epochs = 20
batch_size = 32
#if running with nohup
nohup = True


[testing settings]
#Choose the model to test: best==epoch with min loss, last==last epoch
best_last = best
#number of full images for the test (max 10)
full_images_to_test = 10
#How many original-groundTruth-prediction images are visualized in each image
N_group_visual = 1
#Compute average in the prediction, improve results but require more patches to be predicted
average_mode = True
#Only if average_mode==True. Stride for patch extraction, lower value require more patches to be predicted
stride_height = 46
stride_width = 46
#if running with nohup
nohup = True
```

FIGURE 13: STRUCTURE FILE 2

```
Area under the ROC curve: 0.7024789612104095
Area under Precision-Recall curve: 0.29859295758516197
Jaccard similarity score: 0.8718904135185923
F1 score (F-measure): 0.0

Confusion matrix:[[1978417       0]
 [ 290695       0]]
ACCURACY: 0.8718904135185923
SENSITIVITY: 0.0
SPECIFICITY: 1.0
PRECISION: 0
```

FIGURE 14: PERFORMANCE OF THE MODEL

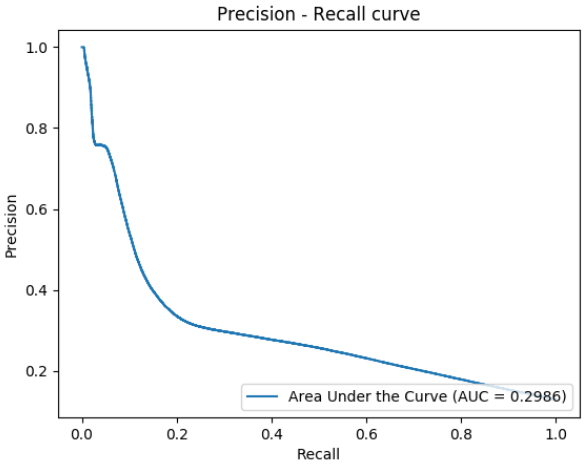

FIGURE 15: PRECISION RECALL

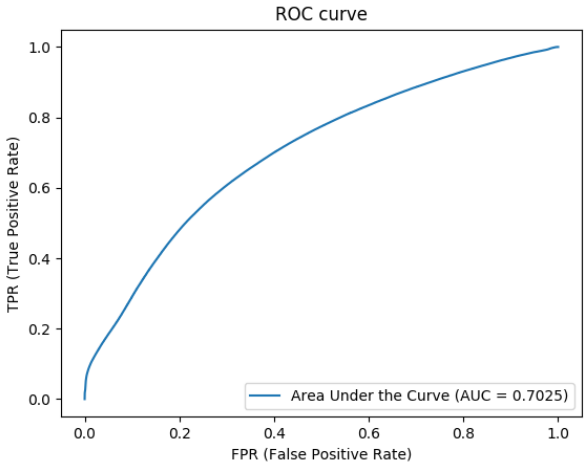

FIGURE 16: ROC CURVE

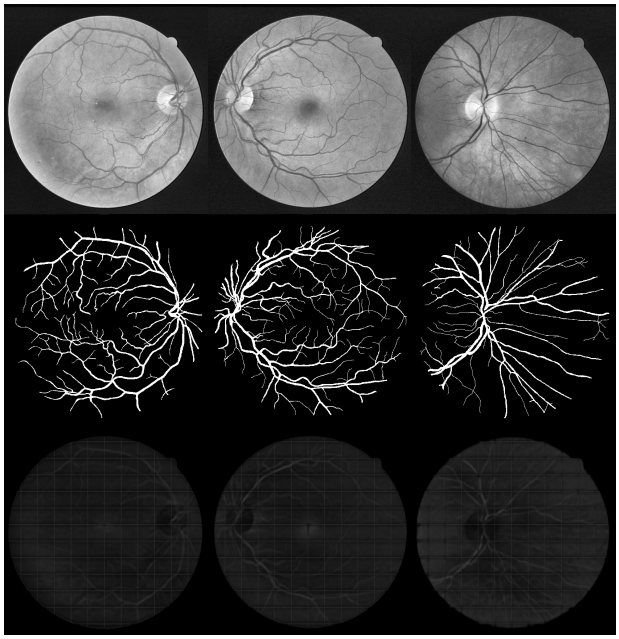

FIGURE 17: PRDECTED SEGMENTATON

Above are 3 of the system-generated blood vessel segmentations. Originals are in the first row, manual segmentations are in the second row, and the system-generated segmentations are in the third row.

FINAL EXPERIMENT

The final experiment ran for 20 hours (please see *figure 18* for configuration of the third experiment). Please observe the results mentioned below.

As you can see, the produced results of the final experiment are better than previous experiments'.

```
[data paths]
path_local =  ./DRIVE_datasets_training_testing/
train_imgs_original = DRIVE_dataset_imgs_train.hdf5
train_groundTruth = DRIVE_dataset_groundTruth_train.hdf5
train_border_masks = DRIVE_dataset_borderMasks_train.hdf5
test_imgs_original = DRIVE_dataset_imgs_test.hdf5
test_groundTruth = DRIVE_dataset_groundTruth_test.hdf5
test_border_masks = DRIVE_dataset_borderMasks_test.hdf5


[experiment name]
name = ANN_NEW_Results


[data attributes]
#Dimensions of the patches extracted from the full images
patch_height = 48
patch_width = 48


[training settings]
#number of total patches:
N_subimgs = 190000
#if patches are extracted only inside the field of view:
inside_FOV = False
#Number of training epochs
N_epochs = 150
batch_size = 32
#if running with nohup
nohup = False


[testing settings]
#Choose the model to test: best==epoch with min loss, last==last epoch
best_last = best
#number of full images for the test (max 20)
full_images_to_test = 20
#How many original-groundTruth-prediction images are visualized in each image
N_group_visual = 1
#Compute average in the prediction, improve results but require more patches to be predicted
average_mode = True
#Only if average_mode==True. Stride for patch extraction, lower value require more patches to be predicted
stride_height = 5
stride_width = 5
#if running with nohup
nohup = False
```

FIGURE 18: STRUCTURE FILE 3

```
Area under the ROC curve: 0.97904342123
Area under Precision-Recall curve: 0.91033534256
Jaccard similarity score: 0.955927459326
F1 score (F-measure): 0.816022456728

Confusion matrix:[[3892134   65128]
 [ 134311  443119]]
ACCURACY: 0.955983977246
SENSITIVITY: 0.767137103303
SPECIFICITY: 0.98352749132
PRECISION: 0.871673209431
```

FIGURE 19: FINAL EXPERIMENT'S PERFORMANCE

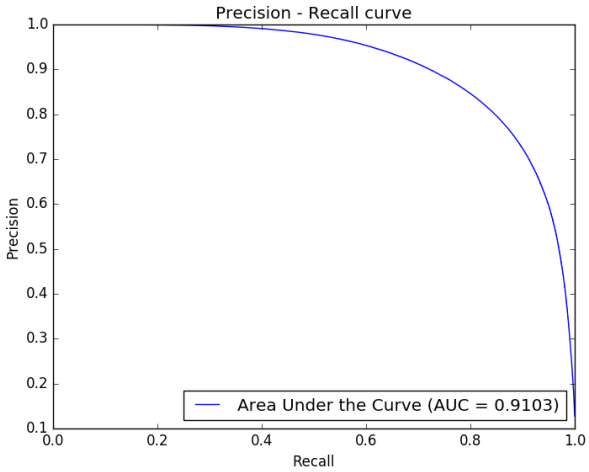

FIGURE 20: FINAL EXPERIMENT'S PRECISION RECALL CURVE

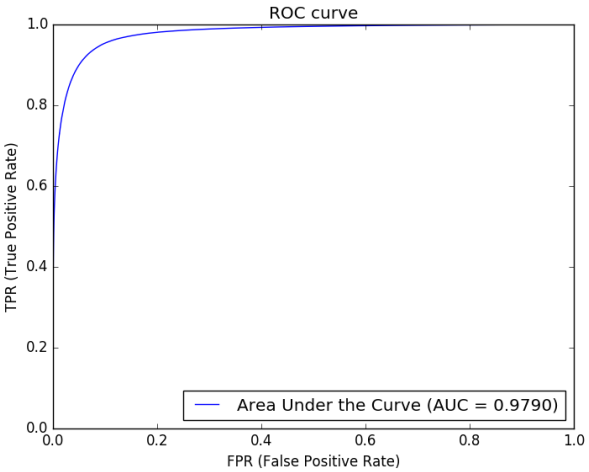

FIGURE 21: FINAL EXPERIMENT'S ROC CURVE

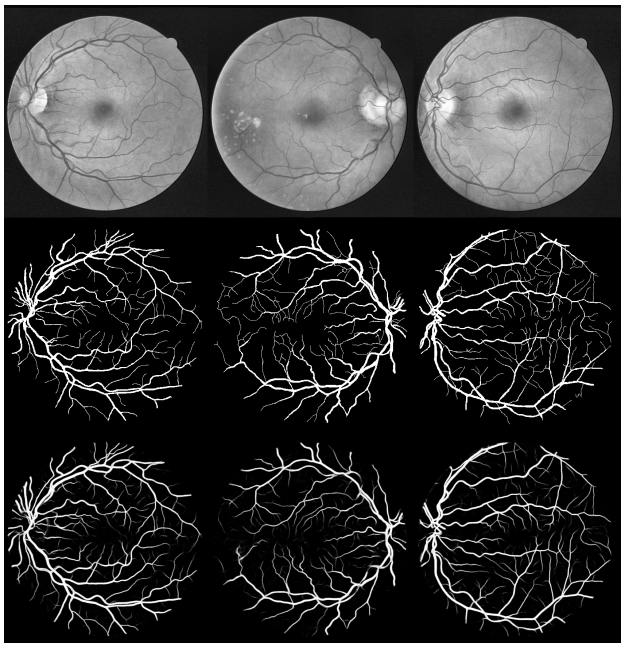

FIGURE 22: FINAL EXPERIMENT'S SYSTEM-GENERATED SEGMENTATION

Above are 3 of the system-generated blood vessel segmentations.

Originals are in the first row, manual segmentations are in the second row, and the system-generated segmentations are in the third row.

Discussion and Conclusion

In this paper, the Eye Fundus Image Analysis (EFIA) is introduced, an automated approach to the retinal blood vessel segmentation. With the employment of ConvNets, one of the most striking image classification method of Artificial Neural Networks, and its u-net architecture, EFIA is enhanced in such a way that it produces results as good as a human annotator, even better in some situations.

Publically available, DRIVE dataset is determined to be an effective source, it rendered sufficient information for the system to develop an intelligent model.

The initial experiment and model creation drew 20 hours on a non-commercial personal laptop with minimal specifications (1.3 GHz, Intel Core i5). On a once trained and prepared model, the blood vessel segmentation takes merely 20 minutes which is a comparable value as opposed to manual human illustrations which would unmistakably take longer and yet have a greater probability to be inefficient. Considering the above, it is safe to say that the proposed automated medical diagnoses can bring evolution to the medical field with an immeasurable novelty.

References

[1] K. Maninis, J. Pont-Tuset, P. Andrés Arbeláez and L. Van Gool, *Deep Retinal Image Understanding*. los Andes: MICCAI, 2016.

[2] O. Ronneberger, P. Fischer and T. Brox, *U-Net: Convolutional Networks for Biomedical Image Segmentation*. Freiburg: Computer Science Department and BIOSS Centre for Biological Signalling Studies, 2015.

[3] Z. Wang and J. Yang, *Diabetic Retinopathy Detection via Deep Convolutional Networks for Discriminative Localization and Visual Explanation*. San Ramon: GE Global Research, 2017.

[4] S. Mallick and V. Gupta, "Learn OpenCV (C++ / Python)", *Learnopencv.com*. [Online]. Available: https://www.learnopencv.com. [Accessed: 09- Mar- 2018].

[5] "DRIVE: Digital Retinal Images for Vessel Extraction", *Isi.uu.nl*. [Online]. Available: https://www.isi.uu.nl/Research/Databases/DRIVE/. [Accessed: 09- Mar- 2018].

[6] "Welcome to OpenCV-Python Tutorials' documentation! — OpenCV-Python Tutorials 1 documentation", *Opencv-python-tutroals.readthedocs.io*, 2018. [Online]. Available: https://opencv-python-tutroals.readthedocs.io/en/latest/. [Accessed: 18- Mar- 2018].

[7] "Convolutional Neural Networks | TensorFlow", *TensorFlow*. [Online]. Available: https://www.tensorflow.org/tutorials/deep_cnn. [Accessed: 24- Mar- 2018].

[8] "Writing your own Keras layers - Keras Documentation", *Keras.io*. [Online]. Available: https://keras.io/layers/writing-your-own-keras-layers/. [Accessed: 28- Mar- 2018].

[9] "Introduction to HDF5", *Support.hdfgroup.org*. [Online]. Available: https://support.hdfgroup.org/HDF5/doc/H5.intro.html. [Accessed: 28- Mar- 2018].

[10] "API Reference — scikit-learn 0.19.1 documentation", *Scikit-learn.org*. [Online]. Available: http://scikit-learn.org/stable/modules/classes.html. [Accessed: 28- Mar- 2018].

[11] W. Rawat and Z. Wang, *Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review*. Florida: Department of Electrical and Mining Engineering, University of South Africa.

[12] "Convolutional Neural Network", *Uk.mathworks.com*. [Online]. Available: https://uk.mathworks.com/discovery/convolutional-neural-network.html. [Accessed: 04- Apr- 2018].

LIST OF FIGURES

APPENDICES

APPENDIX A

```python
import h5py
import numpy as np
from PIL import Image
from matplotlib import pyplot as plt

def load_hdf5(infile):
  with h5py.File(infile,"r") as f:  #"with" close the file after its nested commands
    return f["image"][()]

def write_hdf5(arr,outfile):
  with h5py.File(outfile,"w") as f:
    f.create_dataset("image", data=arr, dtype=arr.dtype)

#convert RGB image in black and white
def rgb2gray(rgb):
    assert (len(rgb.shape)==4)  #4D arrays
    assert (rgb.shape[1]==3)
    bn_imgs = rgb[:,0,:,:]*0.299 + rgb[:,1,:,:]*0.587 + rgb[:,2,:,:]*0.114
    bn_imgs = np.reshape(bn_imgs,(rgb.shape[0],1,rgb.shape[2],rgb.shape[3]))
    return bn_imgs

#group a set of images row per columns
def group_images(data,per_row):
    assert data.shape[0]%per_row==0
    assert (data.shape[1]==1 or data.shape[1]==3)
    data = np.transpose(data,(0,2,3,1))  #corect format for imshow
    all_stripe = []
    for i in range(int(data.shape[0]/per_row)):
        stripe = data[i*per_row]
        for k in range(i*per_row+1, i*per_row+per_row):
            stripe = np.concatenate((stripe,data[k]),axis=1)
        all_stripe.append(stripe)
    totimg = all_stripe[0]
    for i in range(1,len(all_stripe)):
        totimg = np.concatenate((totimg,all_stripe[i]),axis=0)
    return totimg


#visualize image (as PIL image, NOT as matplotlib!)
def visualize(data,filename):
    assert (len(data.shape)==3) #height*width*channels
    img = None
    if data.shape[2]==1:  #in case it is black and white
        data = np.reshape(data,(data.shape[0],data.shape[1]))
    if np.max(data)>1:
        img = Image.fromarray(data.astype(np.uint8))   #the image is already 0-255
    else:
        img = Image.fromarray((data*255).astype(np.uint8))  #the image is between 0-1
    img.save(filename + '.png')
    return img
```

FIGURE 23: HELP FUNCTION FILE 1

```python
#prepare the mask in the right shape for the Unet
def masks_Unet(masks):
    assert (len(masks.shape)==4)  #4D arrays
    assert (masks.shape[1]==1 )  #check the channel is 1
    im_h = masks.shape[2]
    im_w = masks.shape[3]
    masks = np.reshape(masks,(masks.shape[0],im_h*im_w))
    new_masks = np.empty((masks.shape[0],im_h*im_w,2))
    for i in range(masks.shape[0]):
        for j in range(im_h*im_w):
            if  masks[i,j] == 0:
                new_masks[i,j,0]=1
                new_masks[i,j,1]=0
            else:
                new_masks[i,j,0]=0
                new_masks[i,j,1]=1
    return new_masks


def pred_to_imgs(pred, patch_height, patch_width, mode="original"):
    assert (len(pred.shape)==3)  #3D array: (Npatches,height*width,2)
    assert (pred.shape[2]==2 )  #check the classes are 2
    pred_images = np.empty((pred.shape[0],pred.shape[1]))  #(Npatches,height*width)
    if mode=="original":
        for i in range(pred.shape[0]):
            for pix in range(pred.shape[1]):
                pred_images[i,pix]=pred[i,pix,1]
    elif mode=="threshold":
        for i in range(pred.shape[0]):
            for pix in range(pred.shape[1]):
                if pred[i,pix,1]>=0.5:
                    pred_images[i,pix]=1
                else:
                    pred_images[i,pix]=0
    else:
        print("mode " +str(mode) +" not recognized, it can be 'original' or 'threshold'")
        exit()
    pred_images = np.reshape(pred_images,(pred_images.shape[0],1, patch_height, patch_width))
    return pred_images
```

FIGURE 24: HELP FUNCTION FILE 2

```python
############################################
#
#   pre-processing the original images
#
############################################


import numpy as np
from PIL import Image
import cv2

from help_functions import *


#Pre-processing (use for both training and testing!)
def my_PreProc(data):
    assert(len(data.shape)==4)
    assert (data.shape[1]==3)  #Use the original images
    #black-white conversion
    train_imgs = rgb2gray(data)
    #preprocessing:
    train_imgs = dataset_normalized(train_imgs)
    train_imgs = clahe_equalized(train_imgs)
    train_imgs = adjust_gamma(train_imgs, 1.2)
    train_imgs = train_imgs/255.  #reduce to 0-1 range
    return train_imgs


#============================================================
#========= PRE-PROCESSING FUNCTIONS ========================#
#============================================================

#==== histogram equalization
def histo_equalized(imgs):
    assert (len(imgs.shape)==4)  #4D arrays
    assert (imgs.shape[1]==1)  #check the channel is 1
    imgs_equalized = np.empty(imgs.shape)
    for i in range(imgs.shape[0]):
        imgs_equalized[i,0] = cv2.equalizeHist(np.array(imgs[i,0], dtype = np.uint8))
    return imgs_equalized
```

FIGURE 25: PRE-PROCESSING 1

```python

# CLAHE (Contrast Limited Adaptive Histogram Equalization)
#adaptive histogram equalization is used. In this, image is divided into small blocks called "tiles" (tileSize is 8x8 by default in OpenCV). Then each of
    a small area, histogram would confine to a small region (unless there is noise). If noise is there, it will be amplified. To avoid this, contrast limi
    specified contrast limit (by default 40 in OpenCV), those pixels are clipped and distributed uniformly to other bins before applying histogram equaliz
    tile borders, bilinear interpolation is applied
def clahe_equalized(imgs):
    assert (len(imgs.shape)==4)  #4D arrays
    assert (imgs.shape[1]==1)  #check the channel is 1
    #create a CLAHE object (Arguments are optional).
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
    imgs_equalized = np.empty(imgs.shape)
    for i in range(imgs.shape[0]):
        imgs_equalized[i,0] = clahe.apply(np.array(imgs[i,0], dtype = np.uint8))
    return imgs_equalized


# ===== normalize over the dataset
def dataset_normalized(imgs):
    assert (len(imgs.shape)==4)  #4D arrays
    assert (imgs.shape[1]==1)  #check the channel is 1
    imgs_normalized = np.empty(imgs.shape)
    imgs_std = np.std(imgs)
    imgs_mean = np.mean(imgs)
    imgs_normalized = (imgs-imgs_mean)/imgs_std
    for i in range(imgs.shape[0]):
        imgs_normalized[i] = ((imgs_normalized[i] - np.min(imgs_normalized[i])) / (np.max(imgs_normalized[i])-np.min(imgs_normalized[i])))*255
    return imgs_normalized


def adjust_gamma(imgs, gamma=1.0):
    assert (len(imgs.shape)==4)  #4D arrays
    assert (imgs.shape[1]==1)  #check the channel is 1
    # build a lookup table mapping the pixel values [0, 255] to
    # their adjusted gamma values
    invGamma = 1.0 / gamma
    table = np.array([((i / 255.0) ** invGamma) * 255 for i in np.arange(0, 256)]).astype("uint8")
    # apply gamma correction using the lookup table
    new_imgs = np.empty(imgs.shape)
    for i in range(imgs.shape[0]):
        new_imgs[i,0] = cv2.LUT(np.array(imgs[i,0], dtype = np.uint8), table)
    return new_imgs
```

FIGURE 26: PRE-PROCESSING 2

APPENDIX B

```python
import numpy as np
import random
import configparser

from help_functions import load_hdf5
from help_functions import visualize
from help_functions import group_images

from pre_processing import my_PreProc

#To select the same images
# random.seed(10)

#Load the original data and return the extracted patches for training/testing
def get_data_training(DRIVE_train_imgs_original,
                      DRIVE_train_groudTruth,
                      patch_height,
                      patch_width,
                      N_subimgs,
                      inside_FOV):
    train_imgs_original = load_hdf5(DRIVE_train_imgs_original)
    train_masks = load_hdf5(DRIVE_train_groudTruth) #masks always the same
    # visualize(group_images(train_imgs_original[0:20,:,:,:],5),'imgs_train')#.show()  #check original imgs train


    train_imgs = my_PreProc(train_imgs_original)
    train_masks = train_masks/255.

    train_imgs = train_imgs[:,:,9:574,:]  #cut bottom and top so now it is 565*565
    train_masks = train_masks[:,:,9:574,:]  #cut bottom and top so now it is 565*565
    data_consistency_check(train_imgs,train_masks)

    #check masks are within 0-1
    assert(np.min(train_masks)==0 and np.max(train_masks)==1)

    print("\ntrain images/masks shape:")
    print(train_imgs.shape)
    print("train images range (min-max): " +str(np.min(train_imgs)) +' - '+str(np.max(train_imgs)))
    print("train masks are within 0-1\n")

    #extract the TRAINING patches from the full images
    patches_imgs_train, patches_masks_train = extract_random(train_imgs,train_masks,patch_height,patch_width,N_subimgs,inside_FOV)
    data_consistency_check(patches_imgs_train, patches_masks_train)

    print("\ntrain PATCHES images/masks shape:")
    print(patches_imgs_train.shape)
    print("train PATCHES images range (min-max): " +str(np.min(patches_imgs_train)) +' - '+str(np.max(patches_imgs_train)))

    return patches_imgs_train, patches_masks_trainn, patches_imgs_test, patches_masks_test
```

FIGURE 27: PATCH-EXTRACTION 1

```python
#Load the original data and return the extracted patches for training/testing
def get_data_testing(DRIVE_test_imgs_original, DRIVE_test_groudTruth, Imgs_to_test, patch_height, patch_width):
    ### test
    test_imgs_original = load_hdf5(DRIVE_test_imgs_original)
    test_masks = load_hdf5(DRIVE_test_groudTruth)

    test_imgs = my_PreProc(test_imgs_original)
    test_masks = test_masks/255.

    #extend both images and masks so they can be divided exactly by the patches dimensions
    test_imgs = test_imgs[0:Imgs_to_test,:,:,:]
    test_masks = test_masks[0:Imgs_to_test,:,:,:]
    test_imgs = paint_border(test_imgs,patch_height,patch_width)
    test_masks = paint_border(test_masks,patch_height,patch_width)

    data_consistency_check(test_imgs, test_masks)

    #check masks are within 0-1
    assert(np.max(test_masks)==1  and np.min(test_masks)==0)

    print("\ntest images/masks shape:")
    print(test_imgs.shape)
    print("test images range (min-max): " +str(np.min(test_imgs)) +' - '+str(np.max(test_imgs)))
    print("test masks are within 0-1\n")

    #extract the TEST patches from the full images
    patches_imgs_test = extract_ordered(test_imgs,patch_height,patch_width)
    patches_masks_test = extract_ordered(test_masks,patch_height,patch_width)
    data_consistency_check(patches_imgs_test, patches_masks_test)

    print("\ntest PATCHES images/masks shape:")
    print(patches_imgs_test.shape)
    print("test PATCHES images range (min-max): " +str(np.min(patches_imgs_test)) +' - '+str(np.max(patches_imgs_test)))

    return patches_imgs_test, patches_masks_test



# Load the original data and return the extracted patches for testing
# return the ground truth in its original shape
def get_data_testing_overlap(DRIVE_test_imgs_original, DRIVE_test_groudTruth, Imgs_to_test, patch_height, patch_width, stride_height, stride_width):
    ### test
    test_imgs_original = load_hdf5(DRIVE_test_imgs_original)
    test_masks = load_hdf5(DRIVE_test_groudTruth)

    test_imgs = my_PreProc(test_imgs_original)
    test_masks = test_masks/255.
    #extend both images and masks so they can be divided exactly by the patches dimensions
    test_imgs = test_imgs[0:Imgs_to_test,:,:,:]
    test_masks = test_masks[0:Imgs_to_test,:,:,:]
    test_imgs = paint_border_overlap(test_imgs, patch_height, patch_width, stride_height, stride_width)
```

FIGURE 28: PATCH-EXTRACTION 2

```python
    #check masks are within 0-1
    assert(np.max(test_masks)==1  and np.min(test_masks)==0)

    print("\ntest images shape:")
    print(test_imgs.shape)
    print("\ntest mask shape:")
    print(test_masks.shape)
    print("test images range (min-max): " +str(np.min(test_imgs)) +' - '+str(np.max(test_imgs)))
    print("test masks are within 0-1\n")

    #extract the TEST patches from the full images
    patches_imgs_test = extract_ordered_overlap(test_imgs,patch_height,patch_width,stride_height,stride_width)

    print("\ntest PATCHES images shape:")
    print(patches_imgs_test.shape)
    print("test PATCHES images range (min-max): " +str(np.min(patches_imgs_test)) +' - '+str(np.max(patches_imgs_test)))

    return patches_imgs_test, test_imgs.shape[2], test_imgs.shape[3], test_masks


#data consistency check
def data_consistency_check(imgs,masks):
    assert(len(imgs.shape)==len(masks.shape))
    assert(imgs.shape[0]==masks.shape[0])
    assert(imgs.shape[2]==masks.shape[2])
    assert(imgs.shape[3]==masks.shape[3])
    assert(masks.shape[1]==1)
    assert(imgs.shape[1]==1 or imgs.shape[1]==3)
```

FIGURE 29: PATCH-EXTRACTION 3

```python
#extract patches randomly in the full training images
#  -- Inside OR in full image
def extract_random(full_imgs,full_masks, patch_h,patch_w, N_patches, inside=True):
    if (N_patches%full_imgs.shape[0] != 0):
        print("N_patches: plase enter a multiple of 20")
        exit()
    assert (len(full_imgs.shape)==4 and len(full_masks.shape)==4)  #4D arrays
    assert (full_imgs.shape[1]==1 or full_imgs.shape[1]==3)  #check the channel is 1 or 3
    assert (full_masks.shape[1]==1)   #masks only black and white
    assert (full_imgs.shape[2] == full_masks.shape[2] and full_imgs.shape[3] == full_masks.shape[3])
    patches = np.empty((N_patches,full_imgs.shape[1],patch_h,patch_w))
    patches_masks = np.empty((N_patches,full_masks.shape[1],patch_h,patch_w))
    img_h = full_imgs.shape[2]  #height of the full image
    img_w = full_imgs.shape[3] #width of the full image
    # (0,0) in the center of the image
    patch_per_img = int(N_patches/full_imgs.shape[0])  #N_patches equally divided in the full images
    print("patches per full image: " +str(patch_per_img))
    iter_tot = 0   #iter over the total numbe rof patches (N_patches)
    for i in range(full_imgs.shape[0]):  #loop over the full images
        k=0
        while k <patch_per_img:
            x_center = random.randint(0+int(patch_w/2),img_w-int(patch_w/2))
            # print "x_center " +str(x_center)
            y_center = random.randint(0+int(patch_h/2),img_h-int(patch_h/2))
            # print "y_center " +str(y_center)
            #check whether the patch is fully contained in the FOV
            if inside==True:
                if is_patch_inside_FOV(x_center,y_center,img_w,img_h,patch_h)==False:
                    continue
            patch = full_imgs[i,:,y_center-int(patch_h/2):y_center+int(patch_h/2),x_center-int(patch_w/2):x_center+int(patch_w/2)]
            patch_mask = full_masks[i,:,y_center-int(patch_h/2):y_center+int(patch_h/2),x_center-int(patch_w/2):x_center+int(patch_w/2)]
            patches[iter_tot]=patch
            patches_masks[iter_tot]=patch_mask
            iter_tot +=1   #total
            k+=1  #per full_img
    return patches, patches_masks


#check if the patch is fully contained in the FOV
def is_patch_inside_FOV(x_center,y_center,img_w,img_h,patch_h):
    x_ = x - int(img_w/2) # origin (0,0) shifted to image center
    y_ = y - int(img_h/2)  # origin (0,0) shifted to image center
    R_inside = 270 - int(patch_h * np.sqrt(2.0) / 2.0) #radius is 270 (from DRIVE db docs), minus the patch diagonal (assumed it is a square #this is the limit to contain the full patch in the FOV
    radius = np.sqrt((x_*x_)+(y_*y_))
    if radius < R_inside:
        return True
    else:
        return False
```

Figure 30: PATCH-EXTRACTION 4

```python
#Divide all the full_imgs in pachtes
def extract_ordered(full_imgs, patch_h, patch_w):
    assert (len(full_imgs.shape)==4)  #4D arrays
    assert (full_imgs.shape[1]==1 or full_imgs.shape[1]==3)  #check the channel is 1 or 3
    img_h = full_imgs.shape[2]  #height of the full image
    img_w = full_imgs.shape[3] #width of the full image
    N_patches_h = int(img_h/patch_h) #round to lowest int
    if (img_h%patch_h != 0):
        print("warning: " +str(N_patches_h) +" patches in height, with about " +str(img_h%patch_h) +" pixels left over")
    N_patches_w = int(img_w/patch_w) #round to lowest int
    if (img_h%patch_h != 0):
        print("warning: " +str(N_patches_w) +" patches in width, with about " +str(img_w%patch_w) +" pixels left over")
    print("number of patches per image: " +str(N_patches_h*N_patches_w))
    N_patches_tot = (N_patches_h*N_patches_w)*full_imgs.shape[0]
    patches = np.empty((N_patches_tot,full_imgs.shape[1],patch_h,patch_w))

    iter_tot = 0   #iter over the total number of patches (N_patches)
    for i in range(full_imgs.shape[0]):  #loop over the full images
        for h in range(N_patches_h):
            for w in range(N_patches_w):
                patch = full_imgs[i,:,h*patch_h:(h*patch_h)+patch_h,w*patch_w:(w*patch_w)+patch_w]
                patches[iter_tot]=patch
                iter_tot +=1   #total
    assert (iter_tot==N_patches_tot)
    return patches  #array with all the full_imgs divided in patches


def paint_border_overlap(full_imgs, patch_h, patch_w, stride_h, stride_w):
    assert (len(full_imgs.shape)==4)  #4D arrays
    assert (full_imgs.shape[1]==1 or full_imgs.shape[1]==3)  #check the channel is 1 or 3
    img_h = full_imgs.shape[2]  #height of the full image
    img_w = full_imgs.shape[3] #width of the full image
    leftover_h = (img_h-patch_h)%stride_h  #leftover on the h dim
    leftover_w = (img_w-patch_w)%stride_w  #leftover on the w dim
    if (leftover_h != 0):  #change dimension of img_h
        print("\nthe side H is not compatible with the selected stride of " +str(stride_h))
        print("img_h " +str(img_h) + ", patch_h " +str(patch_h) + ", stride_h " +str(stride_h))
        print("(img_h - patch_h) MOD stride_h: " +str(leftover_h))
        print("So the H dim will be padded with additional " +str(stride_h - leftover_h) + " pixels")
        tmp_full_imgs = np.zeros((full_imgs.shape[0],full_imgs.shape[1],img_h+(stride_h-leftover_h),img_w))
        tmp_full_imgs[0:full_imgs.shape[0],0:full_imgs.shape[1],0:img_h,0:img_w] = full_imgs
        full_imgs = tmp_full_imgs
    if (leftover_w != 0):   #change dimension of img_w
        print("the side W is not compatible with the selected stride of " +str(stride_w))
        print("img_w " +str(img_w) + ", patch_w " +str(patch_w) + ", stride_w " +str(stride_w))
        print("(img_w - patch_w) MOD stride_w: " +str(leftover_w))
        print("So the W dim will be padded with additional " +str(stride_w - leftover_w) + " pixels")
        tmp_full_imgs = np.zeros((full_imgs.shape[0],full_imgs.shape[1],full_imgs.shape[2],img_w+(stride_w - leftover_w)))
        tmp_full_imgs[0:full_imgs.shape[0],0:full_imgs.shape[1],0:full_imgs.shape[2],0:img_w] = full_imgs
        full_imgs = tmp_full_imgs
    print("new full images shape: \n" +str(full_imgs.shape))
    return full_imgs
```

Figure 31: PATCH-EXTRACTION 5

```python
#Divide all the full_imgs in pachtes
def extract_ordered_overlap(full_imgs, patch_h, patch_w,stride_h,stride_w):
    assert (len(full_imgs.shape)==4)  #4D arrays
    assert (full_imgs.shape[1]==1 or full_imgs.shape[1]==3)  #check the channel is 1 or 3
    img_h = full_imgs.shape[2]  #height of the full image
    img_w = full_imgs.shape[3] #width of the full image
    assert ((img_h-patch_h)%stride_h==0 and (img_w-patch_w)%stride_w==0)
    N_patches_img = ((img_h-patch_h)//stride_h+1)*((img_w-patch_w)//stride_w+1)  #// --> division between integers
    N_patches_tot = N_patches_img*full_imgs.shape[0]
    print("Number of patches on h : " +str(((img_h-patch_h)//stride_h+1)))
    print("Number of patches on w : " +str(((img_w-patch_w)//stride_w+1)))
    print("number of patches per image: " +str(N_patches_img) +", totally for this dataset: " +str(N_patches_tot))
    patches = np.empty((N_patches_tot,full_imgs.shape[1],patch_h,patch_w))
    iter_tot = 0   #iter over the total number of patches (N_patches)
    for i in range(full_imgs.shape[0]):  #loop over the full images
        for h in range((img_h-patch_h)//stride_h+1):
            for w in range((img_w-patch_w)//stride_w+1):
                patch = full_imgs[i,:,h*stride_h:(h*stride_h)+patch_h,w*stride_w:(w*stride_w)+patch_w]
                patches[iter_tot]=patch
                iter_tot +=1   #total
    assert (iter_tot==N_patches_tot)
    return patches  #array with all the full_imgs divided in patches


def recompone_overlap(preds, img_h, img_w, stride_h, stride_w):
    assert (len(preds.shape)==4)  #4D arrays
    assert (preds.shape[1]==1 or preds.shape[1]==3)  #check the channel is 1 or 3
    patch_h = preds.shape[2]
    patch_w = preds.shape[3]
    N_patches_h = (img_h-patch_h)//stride_h+1
    N_patches_w = (img_w-patch_w)//stride_w+1
    N_patches_img = N_patches_h * N_patches_w
    print("N_patches_h: " +str(N_patches_h))
    print("N_patches_w: " +str(N_patches_w))
    print("N_patches_img: " +str(N_patches_img))
    assert (preds.shape[0]%N_patches_img==0)
    N_full_imgs = preds.shape[0]/N_patches_img
    print("According to the dimension inserted, there are " +str(N_full_imgs) +" full images (of " +str(img_h)+"x" +str(img_w) +" each)")
    full_prob = np.zeros((N_full_imgs,preds.shape[1],img_h,img_w))  #initialize to zero mega array with sum of Probabilities
    full_sum = np.zeros((N_full_imgs,preds.shape[1],img_h,img_w))

    k = 0 #iterator over all the patches
    for i in range(N_full_imgs):
        for h in range((img_h-patch_h)//stride_h+1):
            for w in range((img_w-patch_w)//stride_w+1):
                full_prob[i,:,h*stride_h:(h*stride_h)+patch_h,w*stride_w:(w*stride_w)+patch_w]+=preds[k]
                full_sum[i,:,h*stride_h:(h*stride_h)+patch_h,w*stride_w:(w*stride_w)+patch_w]+=1
                k+=1
    assert(k==preds.shape[0])
    assert(np.min(full_sum)>=1.0)  #at least one
    final_avg = full_prob/full_sum
    print(final_avg.shape)
    assert(np.max(final_avg)<=1.0) #max value for a pixel is 1.0
    assert(np.min(final_avg)>=0.0) #min value for a pixel is 0.0
```

Figure 32: PATCH-EXTRACTION 6

```python
    return final_avg


#Recompone the full images with the patches
def recompone(data,N_h,N_w):
    assert (data.shape[1]==1 or data.shape[1]==3)  #check the channel is 1 or 3
    assert(len(data.shape)==4)
    N_patch_per_img = N_w*N_h
    assert(data.shape[0]%N_patch_per_img == 0)
    N_full_imgs = data.shape[0]/N_patch_per_img
    patch_h = data.shape[2]
    patch_w = data.shape[3]
    N_patch_per_img = N_w*N_h
    #Define and start full recompone
    full_recomp = np.empty((N_full_imgs,data.shape[1],N_h*patch_h,N_w*patch_w))
    k = 0  #iter full img
    s = 0  #iter single patch
    while (s<data.shape[0]):
        #recompone one:
        single_recon = np.empty((data.shape[1],N_h*patch_h,N_w*patch_w))
        for h in range(N_h):
            for w in range(N_w):
                single_recon[:,h*patch_h:(h*patch_h)+patch_h,w*patch_w:(w*patch_w)+patch_w]=data[s]
                s+=1
        full_recomp[k]=single_recon
        k+=1
    assert (k==N_full_imgs)
    return full_recomp


#Extend the full images because patch divison is not exact
def paint_border(data,patch_h,patch_w):
    assert (len(data.shape)==4)  #4D arrays
    assert (data.shape[1]==1 or data.shape[1]==3)  #check the channel is 1 or 3
    img_h=data.shape[2]
    img_w=data.shape[3]
    new_img_h = 0
    new_img_w = 0
    if (img_h%patch_h)==0:
        new_img_h = img_h
    else:
        new_img_h = ((int(img_h)/int(patch_h))+1)*patch_h
    if (img_w%patch_w)==0:
        new_img_w = img_w
    else:
        new_img_w = ((int(img_w)/int(patch_w))+1)*patch_w
    new_data = np.zeros((data.shape[0],data.shape[1],new_img_h,new_img_w))
    new_data[:,:,0:img_h,0:img_w] = data[:,:,:,:]
    return new_data
```

Figure 33: PATCH-EXTRACTION 7

```python
#return only the pixels contained in the FOV, for both images and masks
def pred_only_FOV(data_imgs,data_masks,original_imgs_border_masks):
    assert (len(data_imgs.shape)==4 and len(data_masks.shape)==4)  #4D arrays
    assert (data_imgs.shape[0]==data_masks.shape[0])
    assert (data_imgs.shape[2]==data_masks.shape[2])
    assert (data_imgs.shape[3]==data_masks.shape[3])
    assert (data_imgs.shape[1]==1 and data_masks.shape[1]==1)  #check the channel is 1
    height = data_imgs.shape[2]
    width = data_imgs.shape[3]
    new_pred_imgs = []
    new_pred_masks = []
    for i in range(data_imgs.shape[0]):  #loop over the full images
        for x in range(width):
            for y in range(height):
                if inside_FOV_DRIVE(i,x,y,original_imgs_border_masks)==True:
                    new_pred_imgs.append(data_imgs[i,:,y,x])
                    new_pred_masks.append(data_masks[i,:,y,x])
    new_pred_imgs = np.asarray(new_pred_imgs)
    new_pred_masks = np.asarray(new_pred_masks)
    return new_pred_imgs, new_pred_masks


#function to set to black everything outside the FOV, in a full image
def kill_border(data, original_imgs_border_masks):
    assert (len(data.shape)==4)  #4D arrays
    assert (data.shape[1]==1 or data.shape[1]==3)  #check the channel is 1 or 3
    height = data.shape[2]
    width = data.shape[3]
    for i in range(data.shape[0]):  #loop over the full images
        for x in range(width):
            for y in range(height):
                if inside_FOV_DRIVE(i,x,y,original_imgs_border_masks)==False:
                    data[i,:,y,x]=0.0


def inside_FOV_DRIVE(i, x, y, DRIVE_masks):
    assert (len(DRIVE_masks.shape)==4)  #4D arrays
    assert (DRIVE_masks.shape[1]==1)  #DRIVE masks is black and white
    # DRIVE_masks = DRIVE_masks/255.  #NOOO!! otherwise with float numbers takes forever!!

    if (x >= DRIVE_masks.shape[3] or y >= DRIVE_masks.shape[2]): #my image bigger than the original
        return False

    if (DRIVE_masks[i,0,y,x]>0):  ##== black pixels
        # print DRIVE_masks[i,0,y,x]  #verify it is working right
        return True
    else:
        return False
```

Figure 34: PATCH-EXTRACTION 8

Appendix C

```python
###################################################
#
#   Script to:
#   - Load the images and extract the patches
#   - Define the neural network
#   - Define the training
#
###################################################


import numpy as np
import configparser

from keras.models import Model
from keras.layers import Input, concatenate, Conv2D, MaxPooling2D, UpSampling2D, Reshape, core, Dropout
from keras.optimizers import Adam
from keras.callbacks import ModelCheckpoint, LearningRateScheduler
from keras import backend as K
#from keras.utils.vis_utils import plot_model
from keras.optimizers import SGD

import sys
sys.path.insert(0, './lib/')
from help_functions import *

#function to obtain data for training/testing (validation)
from extract_patches import get_data_training
```

Figure 35: TRAINING 1

```python
#Define the neural network
def get_unet(n_ch,patch_height,patch_width):
    inputs = Input(shape=(n_ch,patch_height,patch_width))
    conv1 = Conv2D(32, (3, 3), activation='relu', padding='same',data_format='channels_first')(inputs)
    conv1 = Dropout(0.2)(conv1)
    conv1 = Conv2D(32, (3, 3), activation='relu', padding='same',data_format='channels_first')(conv1)
    pool1 = MaxPooling2D((2, 2))(conv1)
    #
    conv2 = Conv2D(64, (3, 3), activation='relu', padding='same',data_format='channels_first')(pool1)
    conv2 = Dropout(0.2)(conv2)
    conv2 = Conv2D(64, (3, 3), activation='relu', padding='same',data_format='channels_first')(conv2)
    pool2 = MaxPooling2D((2, 2))(conv2)
    #
    conv3 = Conv2D(128, (3, 3), activation='relu', padding='same',data_format='channels_first')(pool2)
    conv3 = Dropout(0.2)(conv3)
    conv3 = Conv2D(128, (3, 3), activation='relu', padding='same',data_format='channels_first')(conv3)

    up1 = UpSampling2D(size=(2, 2))(conv3)
    up1 = concatenate([conv2,up1],axis=1)
    conv4 = Conv2D(64, (3, 3), activation='relu', padding='same',data_format='channels_first')(up1)
    conv4 = Dropout(0.2)(conv4)
    conv4 = Conv2D(64, (3, 3), activation='relu', padding='same',data_format='channels_first')(conv4)
    #
    up2 = UpSampling2D(size=(2, 2))(conv4)
    up2 = concatenate([conv1,up2], axis=1)
    conv5 = Conv2D(32, (3, 3), activation='relu', padding='same',data_format='channels_first')(up2)
    conv5 = Dropout(0.2)(conv5)
    conv5 = Conv2D(32, (3, 3), activation='relu', padding='same',data_format='channels_first')(conv6)
    #
    conv6 = Conv2D(2, (1, 1), activation='relu',padding='same',data_format='channels_first')(conv5)
    conv6 = core.Reshape((2,patch_height*patch_width))(conv6)
    conv6 = core.Permute((2,1))(conv6)
    ############
    conv7 = core.Activation('softmax')(conv6)

    model = Model(input=inputs, output=conv7)

    # sgd = SGD(lr=0.01, decay=1e-6, momentum=0.3, nesterov=False)
    model.compile(optimizer='sgd', loss='categorical_crossentropy',metrics=['accuracy'])

    return model
```

Figure 36: TRAINING 2

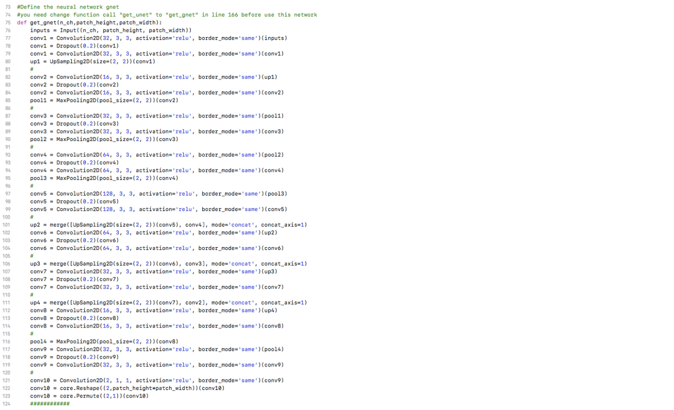

FIGURE 37: TRAINING 3

FIGURE 38: TRAINING 4

FIGURE 39: TRAINING 5

APPENDIX D

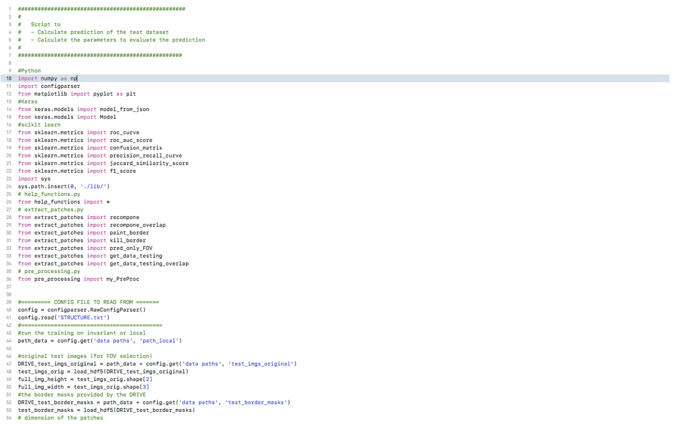

FIGURE 40: TEST 1

FIGURE 41: TEST 2

FIGURE 42: TEST 3

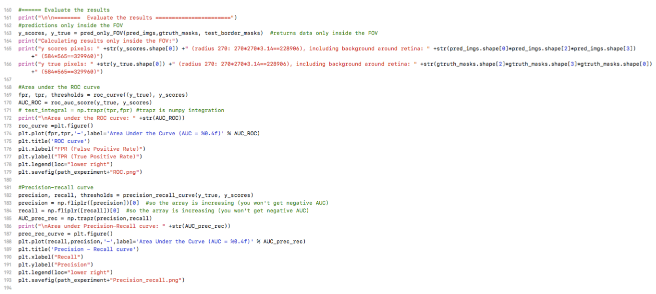

FIGURE 43: TEST 4

FIGURE 44: TEST 5

Appendix D

```
#preparing the hdf5 datasets of the DRIVE database
#============================================================

import os
import h5py
import numpy as np
from PIL import Image



def write_hdf5(arr,outfile):
  with h5py.File(outfile,"w") as f:
    f.create_dataset("image", data=arr, dtype=arr.dtype)


"""--------------PATH TO THE IMAGES----------------------------------------"""

"""--------------train---------------"""

original_imgs_train = "./DRIVE/training/images/"
groundTruth_imgs_train = "./DRIVE/training/1st_manual/"
borderMasks_imgs_train = "./DRIVE/training/mask/"
#test
original_imgs_test = "./DRIVE/test/images/"
groundTruth_imgs_test = "./DRIVE/test/1st_manual/"
borderMasks_imgs_test = "./DRIVE/test/mask/"

"""-----------------------------------"""

Nimgs = 20
channels = 3
height = 584
width = 565
dataset_path = "./DRIVE_datasets_training_testing/"

def get_datasets(imgs_dir,groundTruth_dir,borderMasks_dir,train_test="null"):
    imgs = np.empty((Nimgs,height,width,channels))
    groundTruth = np.empty((Nimgs,height,width))
    border_masks = np.empty((Nimgs,height,width))
    for path, subdirs, files in os.walk(imgs_dir):#list path's directories and files
        for i in range(len(files)):

            #Actual Data
            print("original image: ",files[i])
            img = Image.open(imgs_dir+files[i])
            imgs[i] = np.asarray(img)

            #corresponding ground truth
            groundTruth_name = files[i][0:2] + "_manual1.gif"
            print("ground truth name: ",groundTruth_name)
            g_truth = Image.open(groundTruth_dir + groundTruth_name)
            groundTruth[i] = np.asarray(g_truth)
```

Figure 45: Conversion to HDF5-1

```
            #corresponding border masks
            border_masks_name = ""
            if train_test=="train":
                border_masks_name = files[i][0:2] + "_training_mask.gif"
            elif train_test=="test":
                border_masks_name = files[i][0:2] + "_test_mask.gif"
            else:
                print("specify if train or test!!")
                exit()
            print("border masks name: " + border_masks_name)
            b_mask = Image.open(borderMasks_dir + border_masks_name)
            border_masks[i] = np.asarray(b_mask)

    print("imgs max: ",str(np.max(imgs)))
    print("imgs min: ",str(np.min(imgs)))
    assert(np.max(groundTruth)==255 and np.max(border_masks)==255)
    assert(np.min(groundTruth)==0 and np.min(border_masks)==0)
    print("ground truth and border masks are correctly within pixel value range 0-255 (black-white)")

    #reshaping for my standard tensors
    imgs = np.transpose(imgs,(0,3,1,2))
    assert(imgs.shape == (Nimgs,channels,height,width))
    groundTruth = np.reshape(groundTruth,(Nimgs,1,height,width))
    border_masks = np.reshape(border_masks,(Nimgs,1,height,width))
    assert(groundTruth.shape == (Nimgs,1,height,width))
    assert(border_masks.shape == (Nimgs,1,height,width))
    return imgs, groundTruth, border_masks

if not os.path.exists(dataset_path):
    os.makedirs(dataset_path)

#getting the training datasets
imgs_train, groundTruth_train, border_masks_train = get_datasets(original_imgs_train,groundTruth_imgs_train,borderMasks_imgs_train,"train")
print("saving train datasets")
write_hdf5(imgs_train, dataset_path + "DRIVE_dataset_imgs_train.hdf5")
write_hdf5(groundTruth_train, dataset_path + "DRIVE_dataset_groundTruth_train.hdf5")
write_hdf5(border_masks_train,dataset_path + "DRIVE_dataset_borderMasks_train.hdf5")

#getting the testing datasets
imgs_test, groundTruth_test, border_masks_test = get_datasets(original_imgs_test,groundTruth_imgs_test,borderMasks_imgs_test,"test")
print("saving test datasets")
write_hdf5(imgs_test,dataset_path + "DRIVE_dataset_imgs_test.hdf5")
write_hdf5(groundTruth_test, dataset_path + "DRIVE_dataset_groundTruth_test.hdf5")
write_hdf5(border_masks_test,dataset_path + "DRIVE_dataset_borderMasks_test.hdf5")

```

Figure 46: CONVERSION TO HDF5-2

Code and results are also available on

https://github.com/Huzmorgoth/Aritificial-Neural-Networks