UNIVERSITÉ DE BOURGOGNE

COMPUTER VISION AND ROBOTICS

Using convolutional neural network for the classification of Diabetic retinopathy

---

Utpal Kant
And
Muhammad Zain BASHIR

Centre Universitaire Condorcet - uB, Le Creusot

Introduction

Diabetic retinopathy

Diabetic retinopathy is an eye disease that occurs in people with diabetes and can eventually lead to blindness if left untreated. It almost affects 80% of people having diabetes for more than 20 years. Due to burden of diabetes over the past decades the prevalence of DE is expected to grow exponentially and affect over 300 milion people worldwide by 2025 . Since the progress of the disease is very slow the patient does not experience any symptoms or discomfort until when the disease has already progressed to a later stage. It has been shown that early detection and treatment can reduce the risk of vision loss.

Diabetic retinopathy (DR) is clinically characterized by lesion on the retina which can be detected using fundus photography. The lesions include micro aneurysms, exudates, macular edema and Hemorrhages. Micro aneurysms are tiny blood-filled bulges in the retinal capillaries, exudates are when sugar deposits on the retina when the capillaries become leaky while macular edema is swelling of part of the retina.

Detection

DR is usually detected using Optical coherence tomography or fundus photography. The examiner looks for the clinical characterizations like micro aneurysms, macular edema or hemorrhage and grades the images according to a scale that reflects the extent of retinopathy. Figure 1 shows some the fundus images.

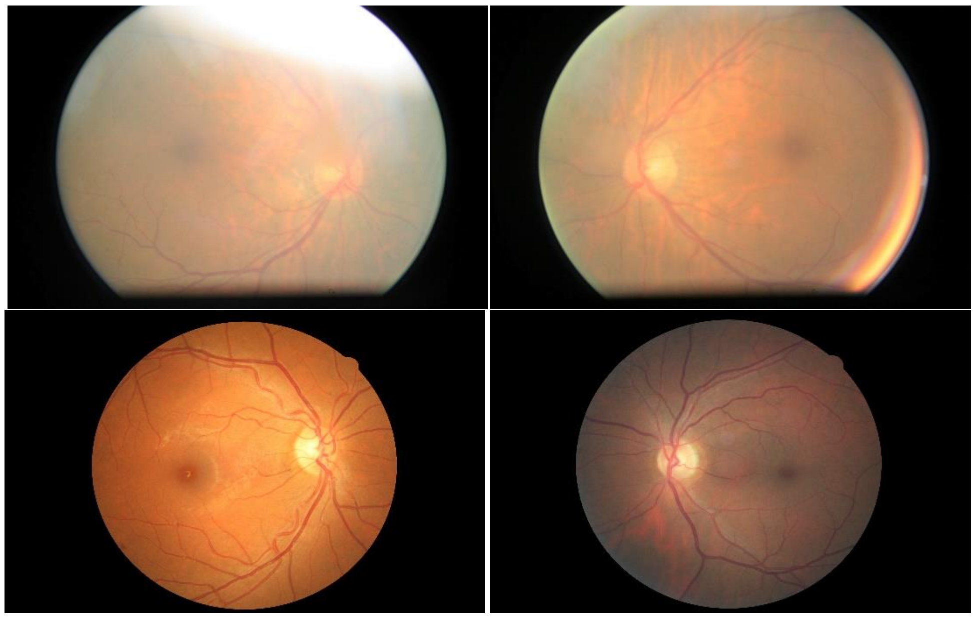


Figure 1: Retinal images as seen in fundus photography

Dataset

The dataset for training our CNN was obtained from Kaggle competition of Diabetic retinopathy detection. The data consists of 54.61 GB of testing images and 32.58 GB of training images to train our classifier. The training images have been labelled according to the extent of retinopathy a patient has by a clinician on a scale of 0-4, according to the following scale:

0 - No DR
1 - Mild
2 - Moderate
3 - Severe
4 - Proliferative DR

The following figure shows an example image for each rating.

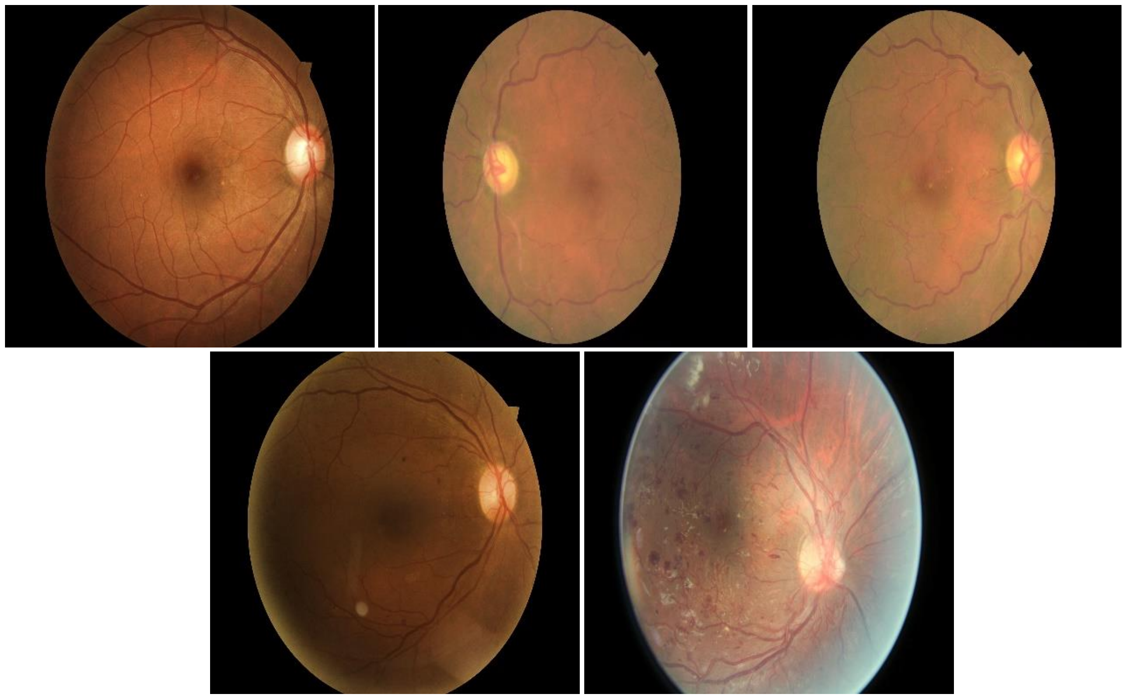


Figure 2: Example image for each rating. (top row =0 1 and 2, bottom row = 3 and 4)

We have 35126 images in the training set as shown in figure 3 thisannotated by a patient id and “left” or “right” (each patient has two images, one per eye) and divided into 5 fairly unbalanced classes (per eye/image, not per patient!)

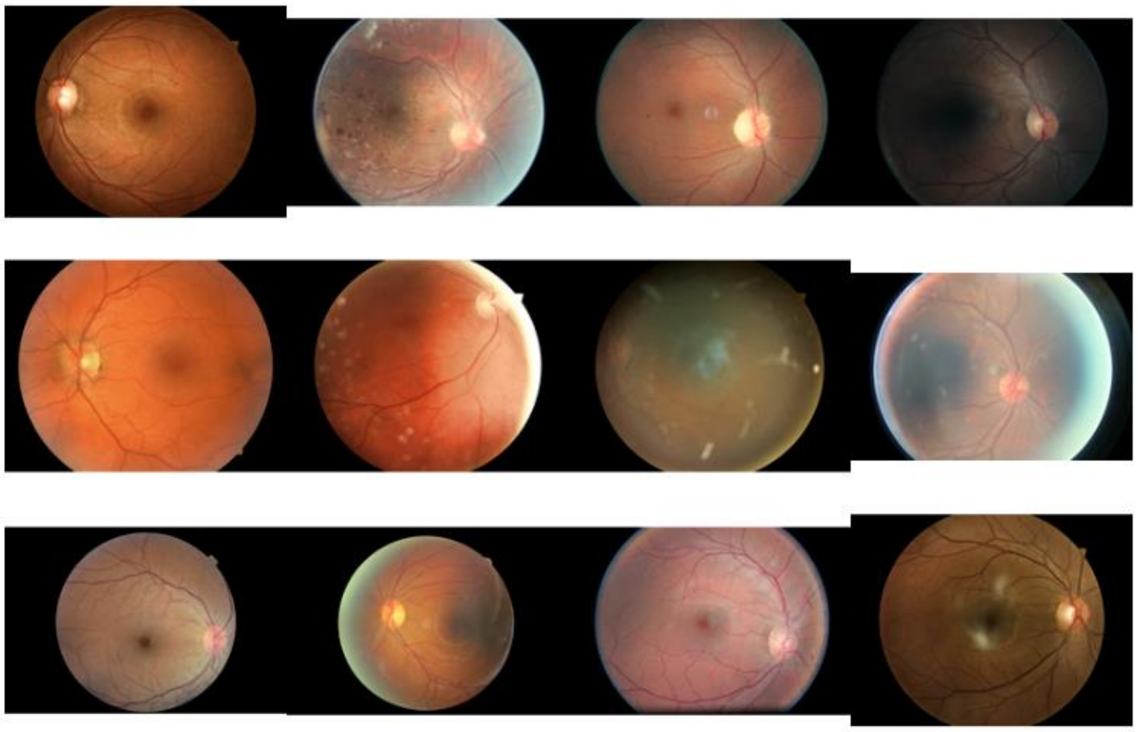


Figure 3 some pseudorandom samples from the training set

Class	Name	Number of images	Percentage
0	Normal	25810	73.48%
1	Mild NPDR	2443	6.96%
2	Moderate NPDR	5292	15.07%
3	Severe NPDR	873	2.48%
4	PDR	708	2.01%

Problem Statement

We have to predict the class (thus, one of the 5 numbers) for each of the **53576 test images** and we implement Convolutional Neural Network to generate the prediction model by training with small subset of training data.

Our Prediction Model

Overview

Our models used a **convolutional network** with the following relatively basic architecture (listing the output size of each layer)

Nr	**Name**	**batch**	**channels**	**width**	**height**	**filter/pool**
0	Input	64	3	512	512	
1	Conv	64	32	256	256	7//2
2	*Max pool*	64	32	127	127	3//2
3	Conv	64	32	127	127	3//1
4	Conv	64	32	127	127	3//1
5	*Max pool*	64	32	63	63	3//2
6	Conv	64	64	63	63	3//1
7	Conv	64	64	63	63	3//1
8	*Max pool*	64	64	31	31	3//2
9	Conv	64	128	31	31	3//1
10	Conv	64	128	31	31	3//1
11	Conv	64	128	31	31	3//1
12	Conv	64	128	31	31	3//1
13	*Max pool*	64	128	15	15	3//2
14	Conv	64	256	15	15	3//1
15	Conv	64	256	15	14	3//1
16	Conv	64	256	15	15	3//1
17	Conv	64	256	15	15	3//1
18	*Max pool*	64	256	7	7	3//2
19	Dropout	64	256	7	7	
20	Maxout (2-pool)	64	512			
21	Concat with image dim	64	514			
22	**Reshape** (merge eyes)	32	1028			
23	Dropout	32	1028			
24	Maxout (2-pool)	32	512			
25	Dropout	32	512			
26	Dense (linear)	32	10			
27	**Reshape** (back to one eye)	64	5			
28	Apply softmax	64	5			

(Where a//b in the last column denotes pool or filter size a x a with stride b x b.)

Software and hardware

We used Python, NumPy and Theano to implement our solution, in combination with the cuDNN library. We also used PyCUDA to implement a few custom kernels. Our code is mostly based on the Lasagne library, which provides a bunch of layer classes and some utilities that make it easier to build neural nets in Theano. We also used opencv and scikit-image for pre-processing and augmentation, and ghalton for quasi-random number generation.

Dependencies:

- Theano: 9a653e3e91c0e38b6643e4452199931e792a24a2
- Lasagne: cf1a23c21666fc0225a05d284134b255e3613335
- Numpy: 1.9.2
- Pandas: 0.15.2
- Scikit-learn: 0.16.0
- Scipy: 0.15.1
- opencv
- IPython: 3.0.0
- Matplotlib: 1.4.2

Pre-processing and data augmentation

We performed very little pre-processing, other than rescaling the images in various ways and then performing global zero mean unit variance (ZMUV) normalization, to improve the stability of training and increase the convergence speed. We experimented with various (combinations of) rescaling strategies. For most networks, we simply rescaled the largest side of each image to a fixed length. We also tried estimating the size of the creatures using image moments.First of all, since the original images are fairly large (say, 3000x2000 pixels on average) and most contained a fairly significant black border, I started by downscaling all the images by a factor of five (without interpolation) and trying to remove most of these black borders.

Data augmentation

We augmented the data to artificially increase the size of the dataset. We used various affine transforms, and gradually increased the intensity of the augmentation as our models started to overfit more. We ended up with some pretty extreme augmentation parameters: The inputs were 512x512 images which were augmented in real-time by

1. *Cropping* with certain probability

2. *Color balance* adjustment
3. *Brightness* adjustment
4. *Contrast* adjustment
5. *Flipping* images with 50% chance
6. *Rotating* images by x degrees, with x an integer in [0, 360[
7. *Zooming* (equal cropping on x and y dimensions)

together with their original image dimensions. Because of the great class imbalance, some classes were **oversampled** to get a more uniform distribution of classes in batches. Somewhere in the middle of training this oversampling stopped and images were sampled from the true training set distribution to

1. Try to control the overfitting, which is particularly difficult when the network sees some images almost ten times more often than others.
2. Have the network fine-tune the predictions on the true class distribution

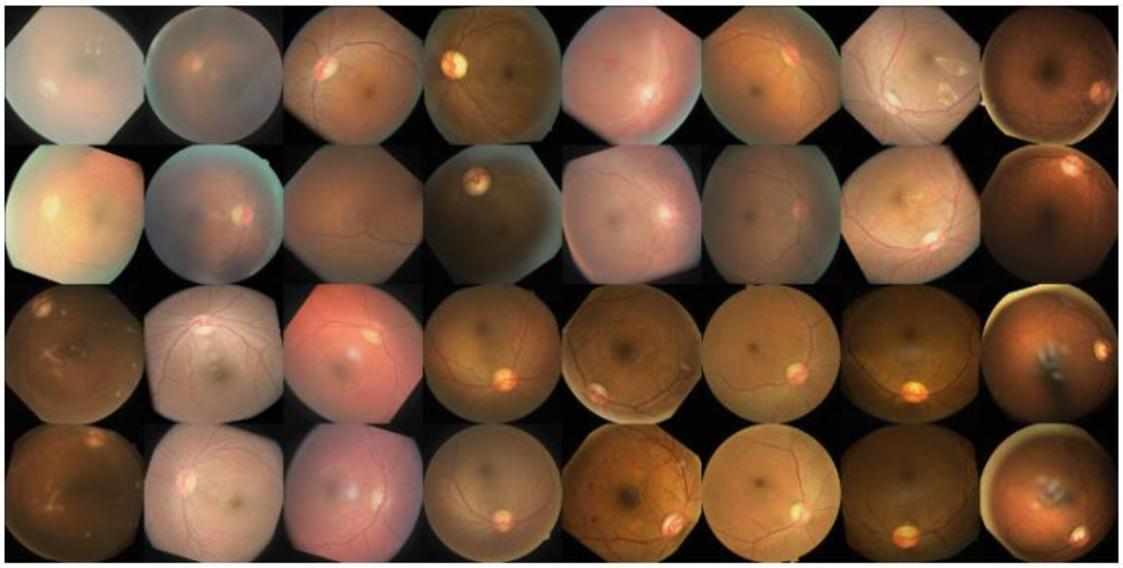


Figure 4 Preprocessed data set

Most of these were implemented from the start. The specific ranges/parameters depend on the model. During training random samples are picked from the training set and transformed before queueing them for input to the network. The augmentations were done by spawning different threads on the CPU such that there was almost no delay in waiting for another batch of samples.

Resizing was done after the cropping and before the other transformations, because it makes some of other operations computationally way too intensive, and can be done in two ways:

1. Rescale while *keeping the aspect ratio* and doing a *centre crop* on the resulting image
2. Normal bilinear rescaling, *destroying the original aspect ratio*

During the training the input is normalised by subtracting the total mean and dividing by the total standard deviation estimated on a few hundred samples before training.The original image dimensions (before they are rescaled to a square input) are also always concatenated as features with the other dense representations.

CNN Model Architecture

Most of our convnet architectures were strongly inspired by OxfordNet: they consist of lots of convolutional layers with 3x3 filters. We used 'same' convolutions (i.e. the output feature maps are the same size as the input feature maps) and overlapping pooling with window size 3 and stride 2. We started with a fairly shallow models by modern standards (~ 6 layers) and gradually added more layers when we noticed it improved performance (it usually did). we were training models with up to 64 layers. The challenge, as always, was balancing improved performance with increased overfitting. We experimented with strided convolutions with 7x7 filters in the first two layers for a while, inspired by the work of He et al., but we were unable to achieve the same performance with this in our networks.

Cyclic pooling

We applied the same stack of convolutional layers to several rotated and flipped versions of the same input image, concatenated the resulting feature representations, and fed those into a stack of dense layers. This allowed the network to use the same feature extraction pipeline to "look at" the input from different angles. Here, we took this a step further. Rather than concatenating the feature representations, we decided to pool across them to get rotation invariance. Here's how it worked in practice: the images in a minibatch occur 4 times, in 4 different orientations. They are processed by the network in parallel, and at the top, the feature maps are pooled together. We decided to call this cyclic pooling, after cyclic groups. The nice thing about 4-way cyclic pooling is that it can be implemented very efficiently: the images are rotated by 0, 90, 180 and 270 degrees. All of these rotations can be achieved simply by transposing and flipping image axes. That means no interpolation is required. Cyclic pooling also allowed us to reduce the batch size by a factor of 4: instead of having batches of 128 images, each batch now contained 32 images and was then turned into a batch with an effective size of 128 again inside the network, by stacking the original batch in 4 orientations. After the pooling step, the batch size was reduced to 32 again. We tried several pooling functions, as well as different positions in the network for the pooling operation (just before the output layer, between hidden layers, …). It turned out that root-mean-square pooling gave much better results than mean pooling or max pooling. We weren’t able to find a good explanation for this, but we suspect it may have something to do with rotational phase invariance.

‘Rolling’ feature maps

Cyclic pooling modestly improved our results, but it can be taken a step further. A cyclic pooling convnet extracts features from input images in four different orientations. An alternative interpretation is that its filters are applied to the input images in four different orientations. That means we can combine the stacks of feature maps from the different orientations into one big stack, and then learn the next layer of features on this combined input. As a result, the network then appears to have 4 times more filters than it actually has! This is cheap to do, since the feature maps are already being computed anyway. We just have to combine them together in the right order and orientation. We named the operation that combines feature maps from different orientations a roll. Roll operations can be inserted after dense layers or after convolutional layers. In the latter case, care has to be taken to rotate the feature maps appropriately, so that they are all aligned. We originally implemented the operations with a few lines of Theano code. This is a nice demonstration of Theano’s effectiveness for rapid prototyping of new ideas. Later on we spent some time implementing CUDA kernels for the roll operations and their gradients, because networks with many rolled layers were getting pretty slow to train. Using your own CUDA kernels with Theano turns out to be relatively easy in combination with PyCUDA. No additional C-code is required. In most of the models we evaluated, we only inserted convolutional roll operations after the pooling layers, because this reduced the size of the feature maps that needed to be copied and stacked together. Note that it is perfectly possible to build a cyclic pooling convnet without any roll operations, but it’s not possible to have roll operations in a network without cyclic pooling. The roll operation is only made possible because the cyclic pooling requires that each input image is processed in four different orientations to begin with.

Nonlinearities

We experimented with various variants of rectified linear units (ReLUs), as well as maxout units (only in the dense layers). We also tried out smooth non-linearities and the ‘parameterized ReLUs’ that were recently introduced by He et al., but found networks with these units to be very prone to overfitting. However, we had great success with (very) leaky ReLUs. Instead of taking the maximum of the input and zero, y = max(x, 0), leaky ReLUs take the maximum of the input and a scaled version of the input, y = max(x, a*x). Here, a is a tunable scale parameter. Setting it to zero yields regular ReLUs, and making it trainable yields parameterized ReLUs. For fairly deep networks (10+ layers), we found that varying this parameter between 0 and 1/2 did not really affect the predictive performance. However, larger values in this range significantly reduced the level of overfitting. This in turn allowed us to scale up our models further. We eventually settled on a = 1/3.

Spatial pooling

We started out using networks with 2 or 3 spatial pooling layers, and we initially had some trouble getting networks with more pooling stages to work well. Most of our final models have 4 pooling stages though. We started out with the traditional approach of 2x2 max-pooling, but eventually switched to 3x3 max-pooling with stride 2 (which we’ll refer to as 3x3s2), mainly because it allowed us to use a larger input size while keeping the same feature map size at the topmost convolutional layer, and without increasing the computational cost significantly. As an example, a network with 80x80 input and 4 2x2 pooling stages will have feature maps of size 5x5 at the topmost convolutional layer. If we use 3x3s2 pooling instead, we can feed 95x95 input and get feature maps with the same 5x5 shape. This improved performance and only slowed down training slightly.

Multiscale architectures

As mentioned before, the images vary widely in size, so we usually rescaled them using the largest dimension of the image as a size estimate. This is clearly suboptimal, because some species of plankton are larger than others. Size carries valuable information. To allow the network to learn this, we experimented with combinations of different rescaling

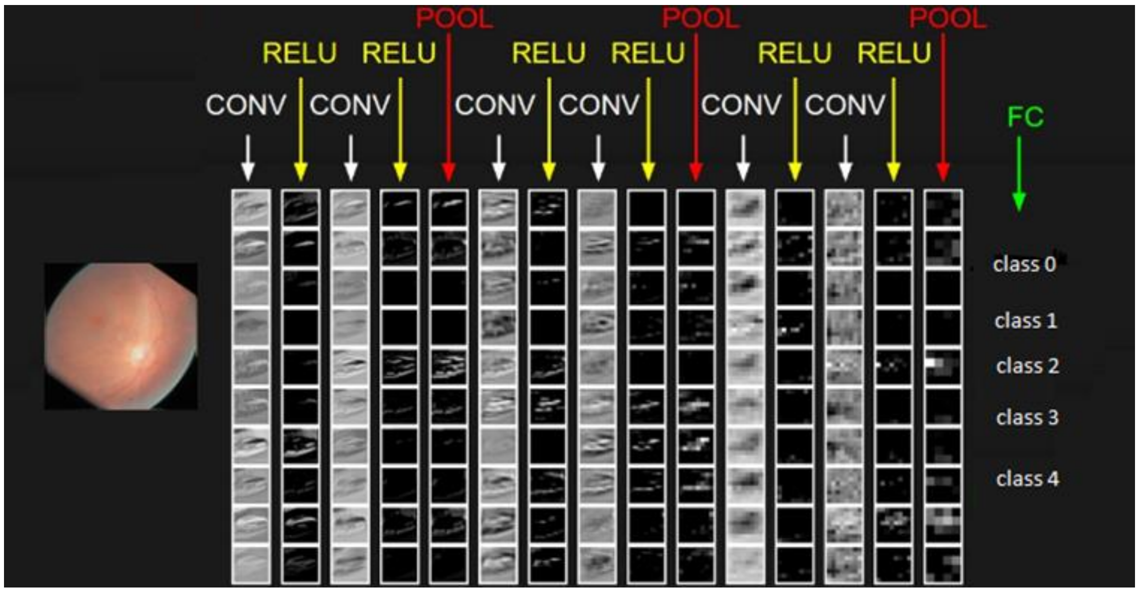


Figure 5 CNN Architecture

strategies within the same network, by combining multiple networks with different rescaled inputs together into ‘multiscale’ networks. What worked best was to combine a network with inputs rescaled based on image size, and a smaller network with inputs rescaled by a fixed factor. Of course this slowed down training quite a bit, but it allowed us to squeeze out a bit more performance.

Additional image features

We experimented with training small neural nets on extracted image features to ‘correct’ the predictions of our convnets. We referred to this as ‘late fusing’ because the feature network and the convnet were joined only at the output layer (before the softmax). We also tried joining them at earlier layers, but consistently found this to work worse, because of overfitting. We thought this could be useful, because the features can be extracted from the raw (i.e. non-rescaled) images, so this procedure could provide additional information that is missed by the convnets. Here are some examples of types of features we evaluated (the ones we ended up using are in bold):

- Image size in pixels
- Size and shape estimates based on image moments
- Hu moments
- Zernike moments

- Parameter Free Threshold Adjacency Statistics
- Linear Binary Patterns
- Haralick texture features
- Features from the competition tutorial
- Combinations of the above

The image size, the features based on image moments and the Haralick texture features were the ones that stood out the most in terms of performance. The features were fed to a neural net with two dense layers of 80 units. The final layer of the model was fused with previously generated predictions of our best convnet-based models. Using this approach, we didn't have to retrain the convnets nor did we have to regenerate predictions (which saved us a lot of time). To deal with variance due to the random weight initialization, we trained each feature network 10 times and blended the copies with uniform weights. This resulted in a consistent validation loss decrease of 0.01 (or 1.81%) on average, which was quite significant near the end of the competition. Interestingly, late fusion with image size and features based on image moments seems to help just as much for multiscale models as for regular convnets. This is a bit counterintuitive: we expected both approaches to help because they could extract information about the size of the creatures, so the obtained performance improvements would overlap. The fact they were fully orthogonal was a nice surprise.

Our CNN architecture

Here's the architecture that works well. It has 64 convolutional layers. This gave us the *basic architecture for 512x512 rescaled input* which was used pretty much until the end:

Nr	**Name**	**batch**	**channels**	**width**	**height**	**filter/pool**
0	Input	64	3	512	512	
1	Conv	64	32	256	256	7//2
2	*Max pool*	64	32	127	127	3//2

Nr	**Name**	**batch**	**channels**	**width**	**height**	**filter/pool**
3	Conv	64	32	127	127	3//1
4	Conv	64	32	127	127	3//1
5	*Max pool*	64	32	63	63	3//2
6	Conv	64	64	63	63	3//1
7	Conv	64	64	63	63	3//1
8	*Max pool*	64	64	31	31	3//2
9	Conv	64	128	31	31	3//1
10	Conv	64	128	31	31	3//1
11	Conv	64	128	31	31	3//1
12	Conv	64	128	31	31	3//1
13	*Max pool*	64	128	15	15	3//2
14	Conv	64	256	15	15	3//1
15	Conv	64	256	15	14	3//1
16	Conv	64	256	15	15	3//1
17	Conv	64	256	15	15	3//1
18	*Max pool*	64	256	7	7	3//2
19	Dropout	64	256	7	7	
20	Maxout (2-pool)	64	512			
21	Concat with image dim	64	514			

Nr	Name	batch	channels	width	height	filter/pool
22	**Reshape** (merge eyes)	32	1028			
23	Dropout	32	1028			
24	Maxout (2-pool)	32	512			
25	Dropout	32	512			
26	Dense (linear)	32	10			
27	**Reshape** (back to one eye)	64	5			
28	Apply softmax	64	5			

(Where `a//b` in the last column denotes pool or filter size `a x a` with stride `b x b`.)

Some things that had also been changed:

1. Using **higher leakiness** on the leaky rectify units, `max(alpha*x, x)`, made a big difference on performance. I started using *alpha=0.5* which worked very well. In the small tests I did, using *alpha=0.3* or lower gave significantly lower scores.
2. Instead of doing the initial downscale with a factor five before processing images, I only downscaled by a factor two. It is unlikely to make a big difference but I was able to handle it computationally so there was not much reason not to.
3. The oversampling of smaller classes was now done with a **resulting uniform distribution of the classes**. But now it also switched back somewhere during the training to the *original* training set distribution. This was done because initially I noticed the distribution of the predicted classes to be quite different from the training set distribution. However, this is not necessarily because of the oversampling (although you would expect it to have a significant effect!) and it appeared to be mostly because of the specific kappa loss optimisation (which takes into account the distributions of the predictions and the ground truth). It is also much more prone to

overfitting when training for a long time on some samples which are 10 times more likely than others.

4. Maxout worked slightly better or at least as well as normal dense layers (but it had fewer parameters).

Experimental Results

5. Firstly We Applied CNN Network in using Tensorflow library in Python on small subset of EyePACS Data Base on i7 Mac.

   Train Data: 150

   Test Data: 90

   Result: Accuracy on test data: 3.99%

3. The Main Problem we faced is hardware requirement to generate the Classification Model. With reference to a paper by Saman Sarraf and Mehdi Ostadhashem. "Big data application in functional magnetic resonance imaging using apache spark", IEEE Future Technologies Conference (FTC), 2016 that Proposed Solution to this problem is Parallel and Distributed Computing Using Hadoop Framework and PySpark

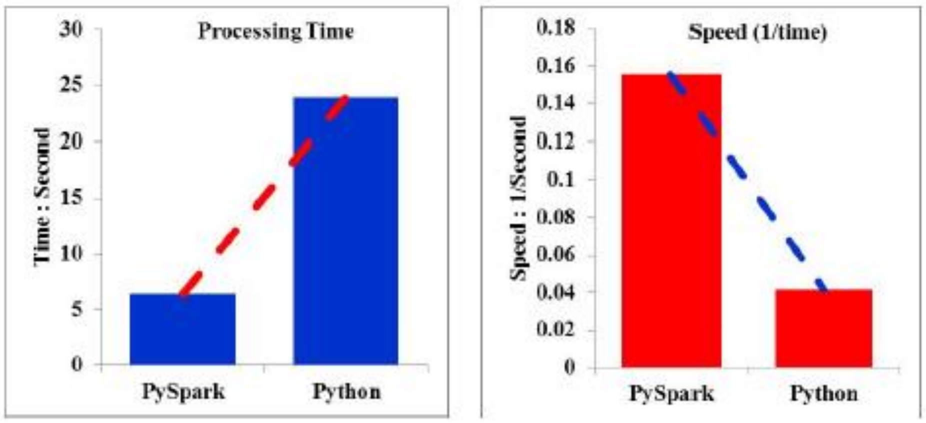
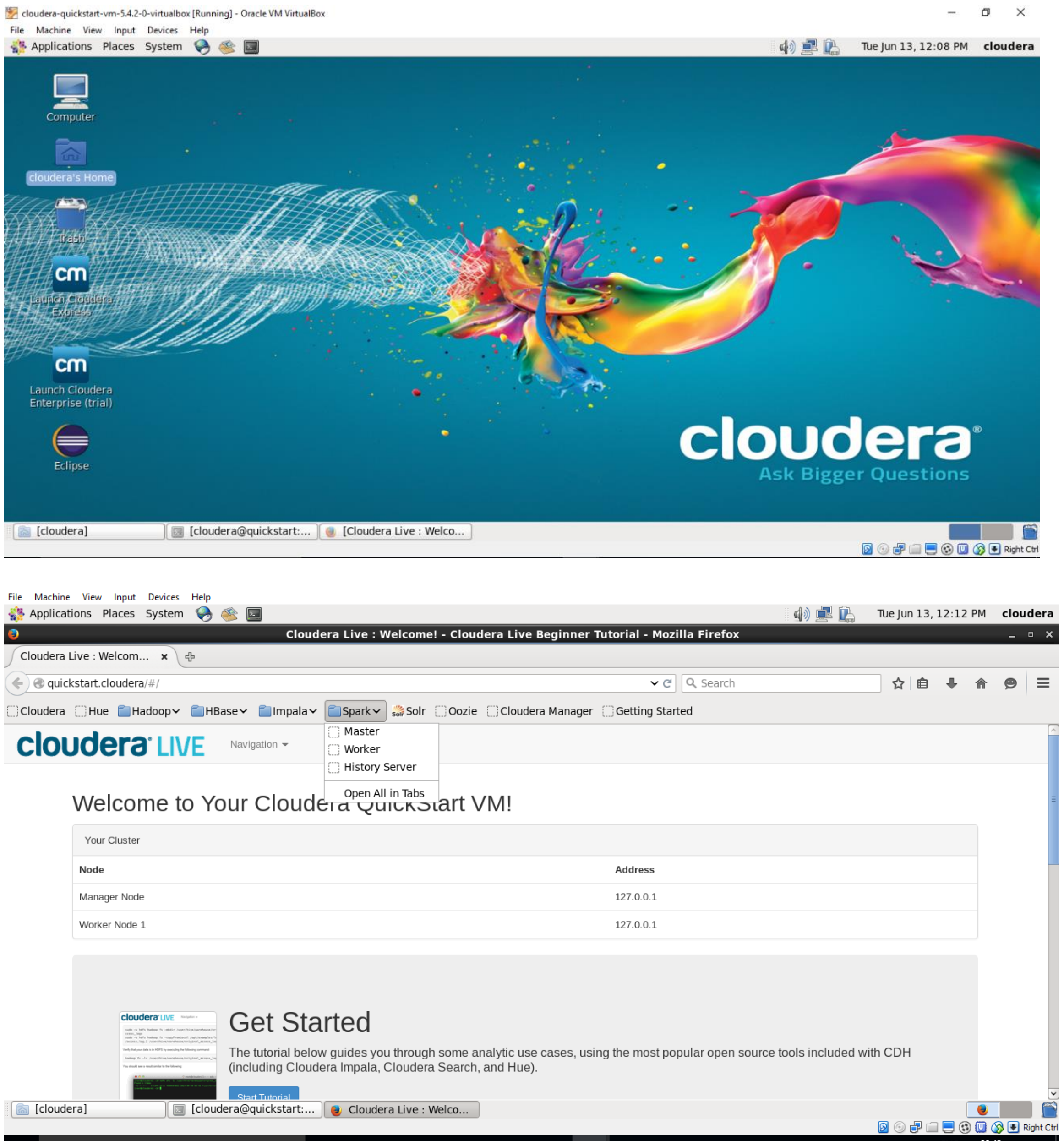


Figure 6: The processing time and speed comparison shows PySpark-based pipeline performs faster

As our result was very poor we installed HADOOP platform on LINUX based CLOUDERA Virtual Machine as shown in figure 7. That enable us to work on large scale data set on web using PySpark.


cloudera-quickstart-vm-5.4.2-0-virtualbox [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Applications Places System
Tue Jun 13, 12:08 PM cloudera
Computer
cloudera's Home
Trash
Launch Cloudera Express
Launch Cloudera Enterprise (trial)
Eclipse
cloudera
Ask Bigger Questions
[cloudera]
[cloudera@quickstart:...
[Cloudera Live : Welco...
Right Ctrl
File Machine View Input Devices Help
Applications Places System
Tue Jun 13, 12:12 PM cloudera
Cloudera Live : Welcome! - Cloudera Live Beginner Tutorial - Mozilla Firefox
Cloudera Live : Welcom...
quickstart.cloudera/#/
Search
Cloudera
Hue
Hadoop
HBase
Impala
Spark
Solr
Oozie
Cloudera Manager
Getting Started
Master
Worker
History Server
Open All in Tabs
cloudera LIVE
Navigation
Start Tutorial
[cloudera]
[cloudera@quickstart:...
Cloudera Live : Welco...
Right Ctrl

Welcome to Your Cloudera QuickStart VM!

Your Cluster

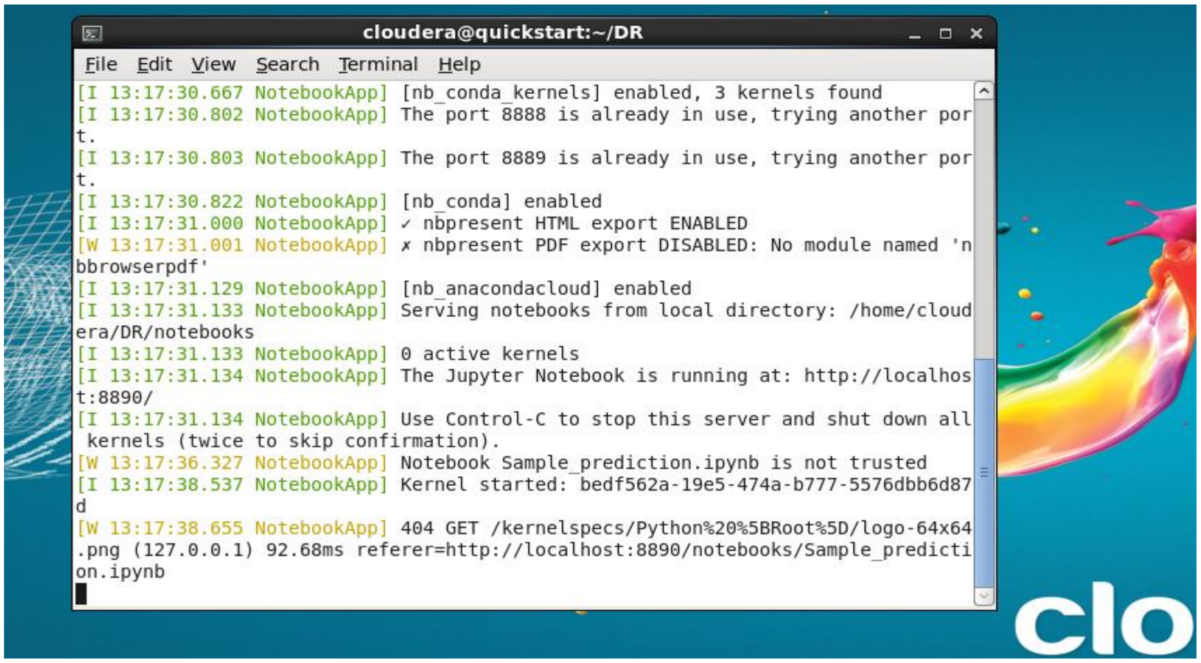
Node	Address
Manager Node	127.0.0.1
Worker Node 1	127.0.0.1

Get Started

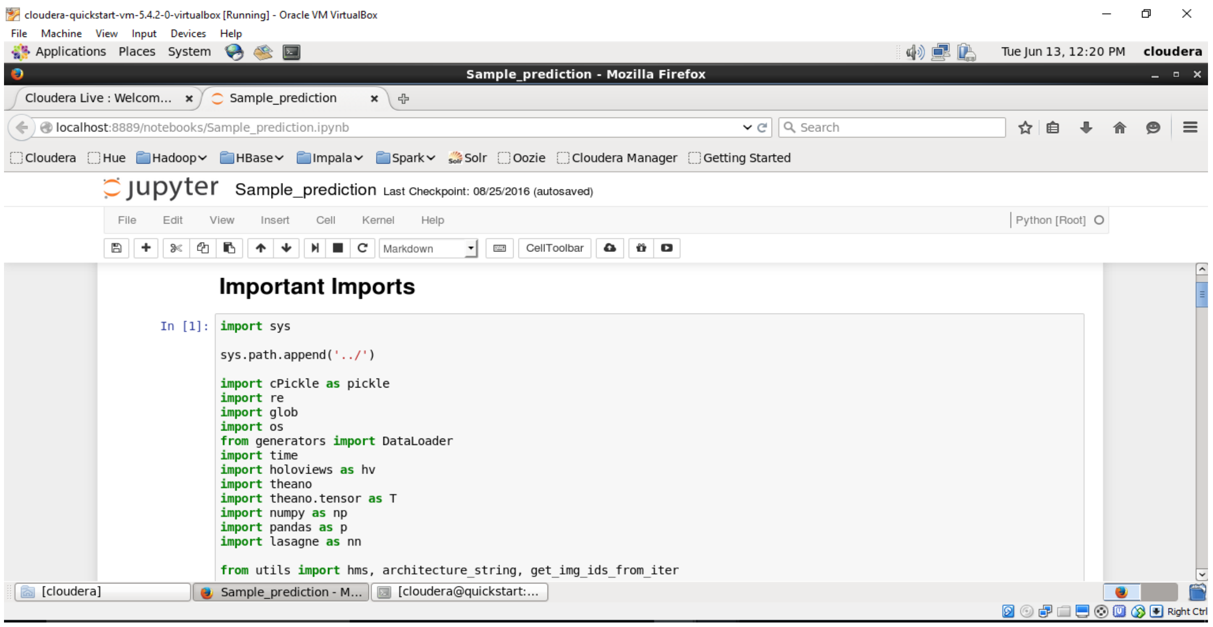
The tutorial below guides you through some analytic use cases, using the most popular open source tools included with CDH (including Cloudera Impala, Cloudera Search, and Hue).

Figure7 : HADOOP Environment

But as we never worked on PySpark before firstly we tried to generate our model using python on the same virtual machine by using 10000 training sample. The training is completed in approximately 10 hours and then we run our model on Jupitor notebook to perform testing and we are getting some good results:



```
[I 13:17:30.667 NotebookApp] [nb_conda_kernels] enabled, 3 kernels found
[I 13:17:30.802 NotebookApp] The port 8888 is already in use, trying another por
t.
[I 13:17:30.803 NotebookApp] The port 8889 is already in use, trying another por
t.
[I 13:17:30.822 NotebookApp] [nb_conda] enabled
[I 13:17:31.000 NotebookApp] ✓ nbpresent HTML export ENABLED
[W 13:17:31.001 NotebookApp] ✗ nbpresent PDF export DISABLED: No module named 'n
bbrowserpdf'
[I 13:17:31.129 NotebookApp] [nb_anacondacloud] enabled
[I 13:17:31.133 NotebookApp] Serving notebooks from local directory: /home/cloud
era/DR/notebooks
[I 13:17:31.133 NotebookApp] 0 active kernels
[I 13:17:31.134 NotebookApp] The Jupyter Notebook is running at: http://localhos
t:8890/
[I 13:17:31.134 NotebookApp] Use Control-C to stop this server and shut down all
 kernels (twice to skip confirmation).
[W 13:17:36.327 NotebookApp] Notebook Sample_prediction.ipynb is not trusted
[I 13:17:38.537 NotebookApp] Kernel started: bedf562a-19e5-474a-b777-5576dbb6d87
d
[W 13:17:38.655 NotebookApp] 404 GET /kernelspecs/Python%20%5BRoot%5D/logo-64x64
.png (127.0.0.1) 92.68ms referer=http://localhost:8890/notebooks/Sample_predicti
on.ipynb
```


cloudera-quickstart-vm-5.4.2-0-virtualbox [Running] - Oracle VM VirtualBox
Sample_prediction - Mozilla Firefox
localhost:8889/notebooks/Sample_prediction.ipynb
Tue Jun 13, 12:20 PM
jupyter Sample_prediction Last Checkpoint: 08/25/2016 (autosaved)
Python [Root]

Important Imports

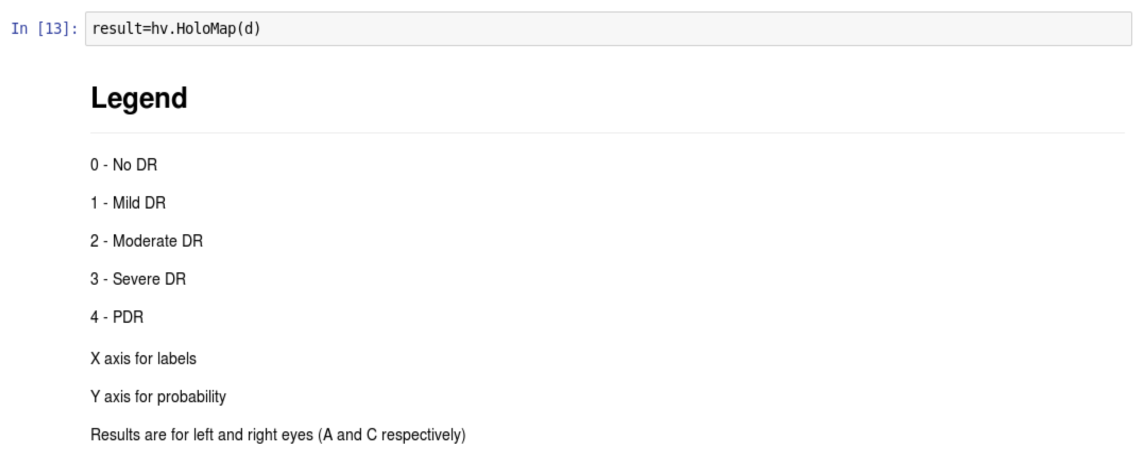
```
In [1]: import sys

        sys.path.append('../')

        import cPickle as pickle
        import re
        import glob
        import os
        from generators import DataLoader
        import time
        import holoviews as hv
        import theano
        import theano.tensor as T
        import numpy as np
        import pandas as p
        import lasagne as nn

        from utils import hms, architecture_string, get_img_ids_from_iter
```

Figure 8: we run our trained model from terminal to open ipython notebook



```
In [13]: result=hv.HoloMap(d)
```

Legend

0 - No DR

1 - Mild DR

2 - Moderate DR

3 - Severe DR

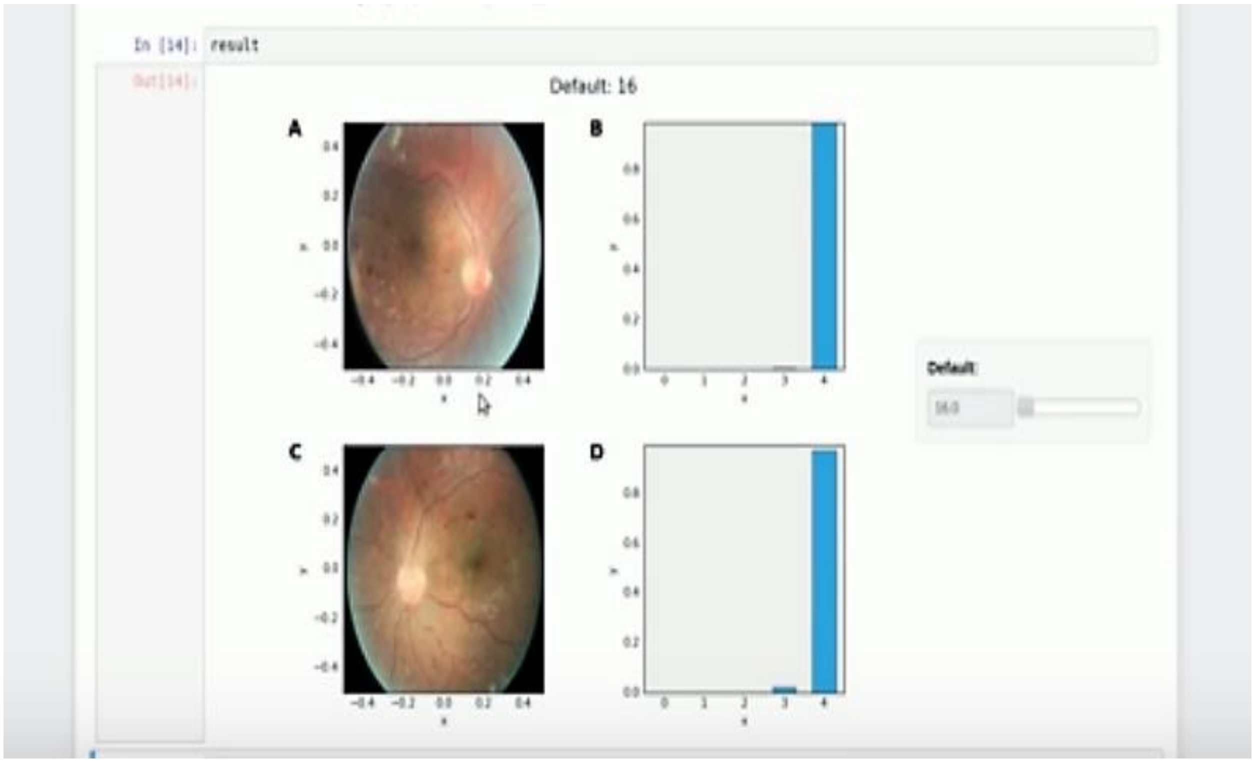
4 - PDR

X axis for labels

Y axis for probability

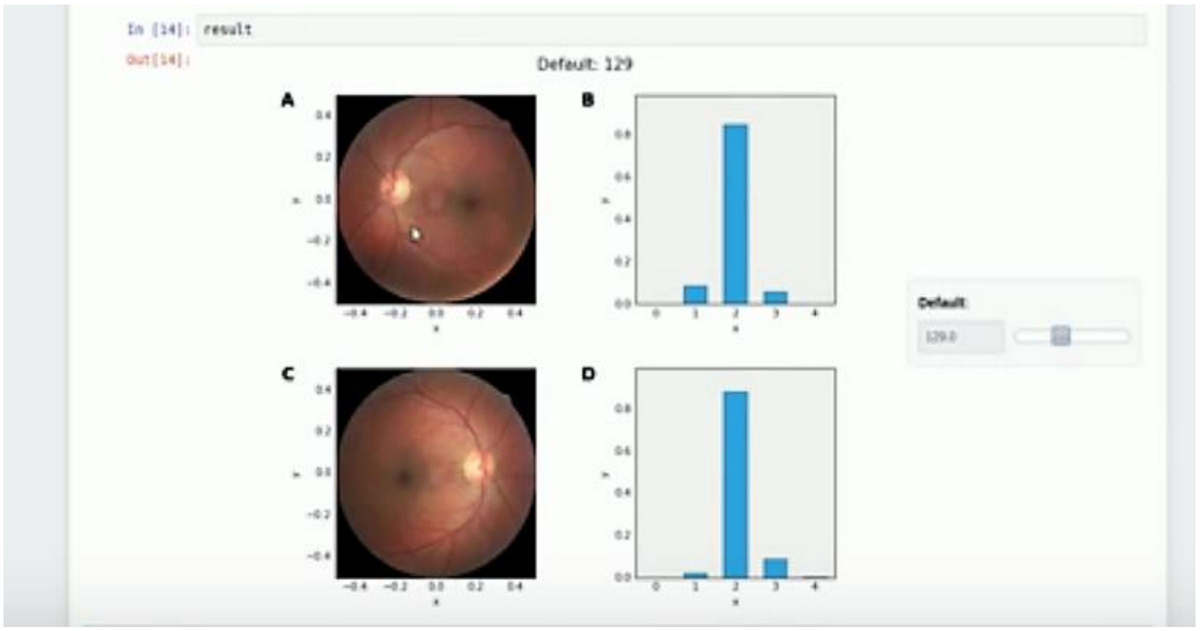
Results are for left and right eyes (A and C respectively)

Figure 9: Result Format



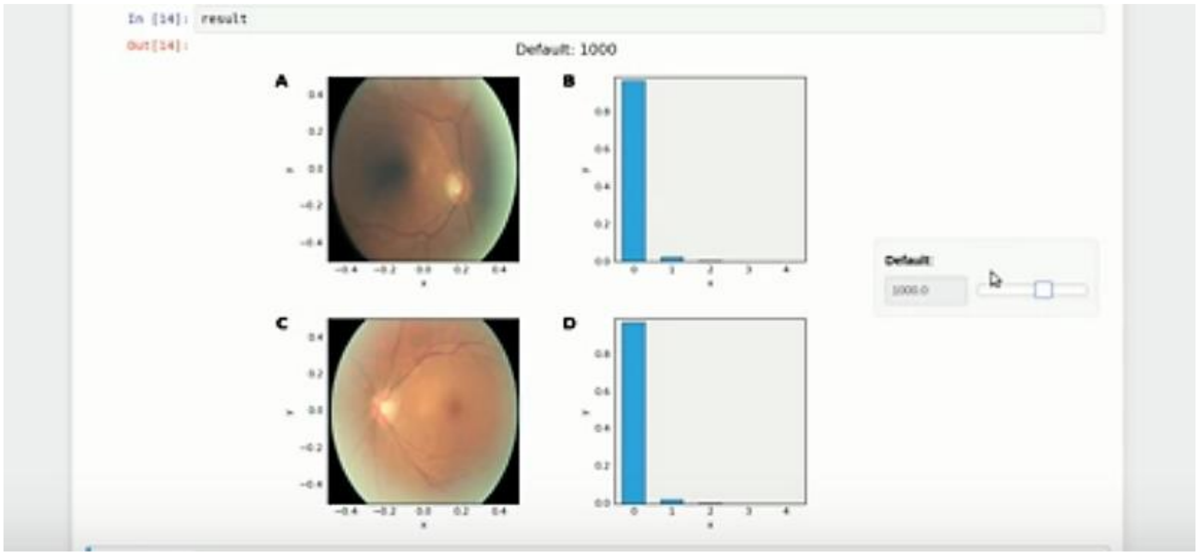
```
In [14]: result
```

Figure 10: Result for an Input Image with level 4 Diabetic Retinopathy



```
In [14]: result
Out[14]:
```

Figure 10: Result for an Input Image with level 2 Diabetic Retinopathy



```
In [14]: result
Out[14]:
```

Figure 10: Result for an Input Image with level 0 Diabetic Retinopathy

So in this experimentation we are getting good results but we have not done the test for accuracy of our model as we are already late for submission of this projects.

Future Plan

We will test the accuracy of our new model and then we will try to train the same model by complete Data Set as well as we will try to generate new prediction model using Pyspark with 5 nodes in a cluster and will try to compare the computation time with python and PySpark.

References

1. Yau JW, Rogers SL, Kawasaki R, Lamoureux EL, Kowalski JW, et al. (2012) Global prevalence and major risk factors of diabetic retinopathy. Diabetes Care 35: 556–564.

2. (1998) Intensive blood-glucose control with sulphonylureas or insulin compared with conventional treatment and risk of complications in patients with type 2 diabetes (UKPDS 33). UK Prospective Diabetes Study (UKPDS) Group. Lancet 352: 837–853.

3. (1995) Progression of retinopathy with intensive versus conventional treatment in the Diabetes Control and Complications Trial. Diabetes Control and Complications Trial Research Group. Ophthalmology 102: 647–661.

4. van Leiden HA, Dekker JM, Moll AC, Nijpels G, Heine RJ, et al. (2002) Blood pressure, lipids, and obesity are associated with retinopathy: the hoorn study. Diabetes Care 25: 1320–1325.

5. Esteves J, Laranjeira AF, Roggia MF, Dalpizol M, Scocco C, et al. (2008) [Diabetic retinopathy risk factors]. Arq Bras Endocrinol Metabol 52: 431–441.

6. Abramoff MD, Niemeijer M, Russell SR (2010) Automated detection of diabetic retinopathy: barriers to translation into clinical practice. Expert Rev Med Devices 7: 287–296.

7. Faust O, Acharya UR, Ng EY, Ng KH, Suri JS (2012) Algorithms for the automated detection of diabetic retinopathy using digital fundus images: a review. J Med Syst 36: 145–157.

8. Priya R, Aruna P (2011) Review of automated diagnosis of diabetic retinopathy using support vector machines. International Journal of Applied Engineering Research 1: 844–862.

9. Abramoff MD, Niemeijer M, Suttorp-Schulten MS, Viergever MA, Russell SR, et al. (2008) Evaluation of a system for automatic detection of diabetic retinopathy from color fundus photographs in a large population of patients with diabetes. Diabetes Care 31: 193–198.

10. Quellec G, Lamard M, Abramoff MD, Decenciere E, Lay B, et al. (2012) A multiple-instance learning framework for diabetic retinopathy screening. Med Image Anal 16: 1228–1240.

11. Quellec G, Lamard M, Cazuguel G, Bekri L, Daccache W, et al. (2011) Automated assessment of diabetic retinopathy severity using content-based image retrieval in multimodal fundus photographs. Invest Ophthalmol Vis Sci 52: 8342–8348.

12. Quellec G, Russell SR, Abramoff MD (2011) Optimal filter framework for automated, instantaneous detection of lesions in retinal images. IEEE Trans Med Imaging 30: 523–533.

13. M. Adala, Désiré Sidibéa, Sharib Alia, Edward Chaumb,Thomas P. Karnowskic, Fabrice Mériaudeauaa "Automated detection of microaneurysms usingscale-adapted blob analysis and semi-supervisedlearning" Elsevier journal of computer methods and programs in biomedicine 114(2014) 1-10.

14. T. Spencer, R. Phillips, P. Sharp, J. Forrester, Automateddetection and quantification of microaneurysms influorescein angiograms, Graefe's Archive for Clinical andExperimental Ophthalmology 230 (1) (1992) 36–41.

15. Saman Sarraf and Mehdi Ostadhashem. "Big data application in functional magnetic resonance imaging using apache spark", IEEE Future Technologies Conference (FTC), 2016.