Università degli Studi di Padova

Scuola di Ingegneria

Corso di Laurea in

Ingegneria delle Telecomunicazioni

A deep learning approach to the analysis of retinal images

Laureando

Enrico Vincenzi

Relatore

Prof. Enrico Grisan

Co-relatore

Prof. Emanuele Trucco

Anno Accademico 2016/2017

A ...

Quote

Author

Contents

Abstract

Motivation
This work is focused on the possible applications of Deep Learning in retinal fundus images analysis. Deep Learning is an advanced machine learning technique that is revolutionizing all data-based disciplines with unheard-of performances in signal analysis. In particular Artificial Intelligence, Computer Vision and Image Analysis are benefiting from huge improvement in image and sound classification, segmentation and pattern recognition. This progress in these fields is extendible to all medical fields where images or general signals are used. It is possible to exploit deep learning to improve disease diagnosis? Is possible to take advantage of AI features of Deep Learning to improve and speed up malady investigation?

Aim and Methods
In this project we will try to give an answer using Deep Learning early to classify diabetic retinopathy and then trying to visualise the salient regions inside fundus images used for the classification. In this way is possible to see how a deep neural network discriminates between different classes and if this information can be useful for early diagnosis or biomarkers discovery. First, Deep Learning and neural networks theory is ; a literature review on the state of the art is eventually presented and software environment and set-up is discussed. Secondly different retinal datasets with artificial markers are shown and used to test and study Grad-CAM visualisation technique. Finally results and conclusions are treated.

Results
Each dataset with different kind of artificial markers give a different result more or less successful. This seems very dependent on the kind of filters learned by the network during the training step.

Conclusion
Deep Learning capabilities have been confirmed in this work, but a lot is necessary to do to understand better how to exploit the information of the trained networks especially for particular images as the medical ones are. The current state of the art is not enough and much more can be achieved improving the way how network filters are obtained and used.

Chapter 1

Introduction

Deep learning is a machine learning technique that, using deep neural networks, is being used in more and more fields. The performances reached with signals classification, segmentation, elaboration are changing Image analysis, computer vision and machine learning itself. In particular medical image analysis is a field where Deep Learning can improve the quality and speed of research ([?]). In particular fundus retinal image analysis is a field where an improvement of diagnosis precision and biomarker discovery can mean a good improvement on medicine quality. In fact roughly speaking, fundus images are just photos of patient retina and, therefore, can be obtained with non invasive methods. Indeed, a lot of work and research is involving Deep Learning and retinal images ([1], [2], [3], [4], [5], [6], [7], [8]). The aim of this work is understand better how is possible to exploit the information gathered by Deep Learning to use it not only for classifying different disease, but to find something potential new that can help the diagnosis. The idea is then to visualise the important regions used by Deep Learning to classify and understand more precisely what a network use to get good performances. Therefore, the analysis of visualisation techniques become the main subject in this work. The technique used are:

- Filter visualisation.
- Grad-CAM: Gradient-weighted Class Activation Mapping ([30], [31]).

The first approach is the simplest and consist in visualising the filters learned by the neural network during training. These filters are involved in the second technique; visualise the filters can be important both to understand which are the most important features of the images and to understand better the output of Grad-CAM. Grad-CAM, instead, is the state-of-the-art in important region visualisation. Indeed allows to visualise the region of the image used by the network to classify it correctly. Is then the core of this thesis, because its performances can tell how much is possible to understand from a trained network. The first step is to choose a network architecture that works well with Grad-CAM and has been used for natural images classification in order to have a benchmark. The best Deep Learning network architecture at March 2017 is Kaiming He ResNet ([29]), but, unfortunately, its non sequential architecture is not the best choice for GradCAM technique. Then a VGG16 has been chosen as fixed architecture for all the experiments. Its sequential architecture and wide use in literature are perfect for the aim of this work (that is not the best classification performance possible). The second step is to prepare particular datasets that can help to understand the behave of training of the network and of Grad-CAM. These dataset are simply fundus retinal images to which some artificial markers are printed on. The markers can be texure (grass, granit, fabric) blobs or non-medical objects as cars, animals, etc. The different result obtained with different artificial markers could help to understand better Grad-CAM performances. For each dataset the VGG16 hare trained in order to obtain a 100% classification accuracy for each of the artificial markers datasets (100% accuracy seems too much but is possible because Deep Learning achieve the best performances in this conditions). The idea is then to reduce as much as possible the number of variables that can influence Grad-CAM and see its strength and weak points.

To achieve these performances and study all that can be taken in account for Grad-CAM is necessary go through Deep Learning Theory and Grad-CAM algorithm.

Chapter 2

Deep Learning theory

In this chapter will be explored what Deep Learning is and what fundamental features will be exploited for visualisation.

2.1 Introduction to machine learning

Machine learning is a core sub-field of Artificial Intelligence. The aim of Machine learning is study flexible computer algorithms that are able to learn. This learning that is being done is always based on some kind of experience based on known input data or instructions.

> **Definition:** *A computer program is said to* ***learn*** *from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.*

Tom M. Mitchell [9]

Therefore, in general learning is about improving future performance using past experience, reducing as more as possible human intervention or assistance. In general Machine learning paradigm can be viewed as "programming by demonstration", where the approach emphasises working on concrete examples rather than describing an abstract procedure. [10]

Machine learning tasks are usually classified in three different wide categories, depending on the nature of the problem faced[11]:

- **Supervised learning**: the algorithm learning is guided through inputs and their desired outputs given by a "teacher". The goal is to build a rule that maps inputs into their outputs.

- **Unsupervised learning**: the input given is not labeled and the goal of the alghorithm is to infer a function to describe hidden structure or pattern in the input.

- **Reinforcement Learning**: the inputs are a set of feedbacks coming from a dynamic environment that the algorithm is facing. The aim is to perform a predetermined goal (playing a video-game or driving a vehicle).

Deep Learning can be implemented both as supervised or unsupervised technique. The problem faced in this thesis required a supervised approach, therefore only this branch of machine learning will be discussed.

2.2 Supervised Learning

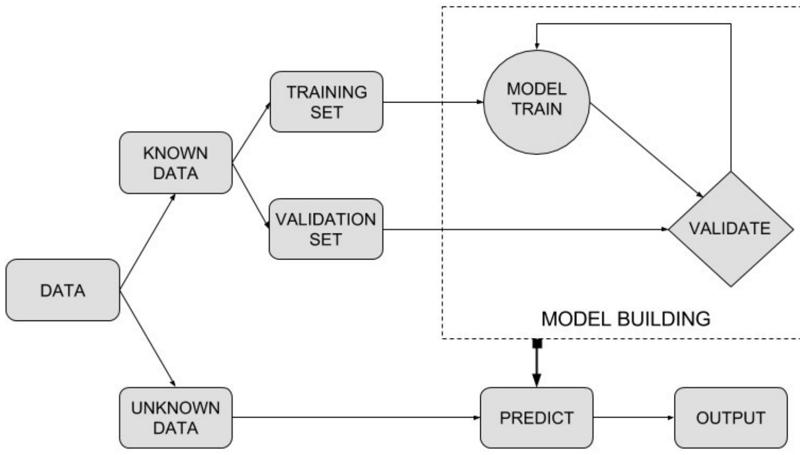


Figure 2.1: Supervised Learning flowchart

As disclosed previously Supervised Learning is an approach to learning that require a known dataset. This set is provided of both inputs and correct outputs for the algorithm used. Starting from this set of examples the program is guided to describe a model able to predict the correct output. At this point the prediction model must be validated with another known dataset independent from the training set. Only when the validation phase is satisfactory the algorithm can be considered reliable for use on unknown data. Therefore, given a supervised problem and the data type, learning steps are:

Algorithm selection The first step is to choose the supervised algorithm to use. Every method has different strength and weak point. The choice depends on the particular problem and on the kind and amount of available data. Some of these algorithms are: Support Vector Machine (SVM), Decision Tree, Artificial Neural Network and Deep Learning. In this work the focus will be on Deep Learning, extension of ANN, for reasons that will be explained in the next sections.

Training The training phase is probably the most important one, as the final performances depend on the predictive model built.

- A known dataset is selected; must be as more representative of the problem as possible. Using dataset not general enough can lead to overfitting and to bad performances. This set, the **training set**, must provide an output (**label**) for each listed input.

- The algorithm is trained with the selected dataset. The aim of this phase is trying to build a model able to fit the data provided, that is predict the correct output for each input provided as best as possible.

Validation The validation phase is important to test the performances achieved by the prediction model built in the previous phase.

- Another known dataset, called **test set**, is prepared. The dataset must provide, as the training set, reliable input and output for each example. An important property of this set is that it should be as independent as possible from the training one.

- The previously trained algorithm is here used to predict the input data of the test set. Only the input are used and the output are predicted by the algorithm and stored. The fundamental difference from the train step is that, in this one, the output label are not used to improve the prediction capabilities of the model, but only to evaluate its performances.

- The predicted outputs are validated using the known outputs. The performance are hence evaluated and analysed. If they are satisfactory it is possible to go to the final step, otherwise the algorithm or the training phase must be reviewed with different precautions or parameters.

Model Deployment Once the algorithm is trained and validated, it is possible to use it as an automatic system to solve the original problem on new data.

2.3 Artificial Neural Networks

Deep Learning is basically an extension of Artificial Neural Networks. Therefore, is important understand this technique before move on.

2.3.1 What is an ANN?

ANN is an information processing paradigm mainly inspired by biological nervous systems. It is composed by a high number of processing units, called neurons, working in unison to solve a specific task. Learning process in ANN involves, like in a biological system, the adjustments of the connections between the processing units.

2.3.2 Perceptron

One of the simplest Artificial Neural Networks algorithms is the Perceptron, introduced in its simplest version by Rosenblatt in 1958, as supervised machine learning algorithm for binary classification. It is partially inspired by the biological neuron, and in some way emulates its behave. This is the reason why this systems are called artificial neural networks.

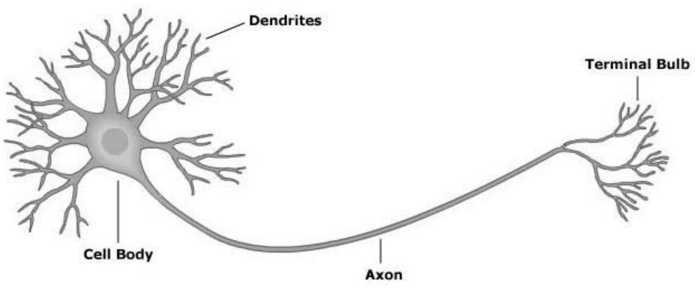


Figure 2.2: Biological neuron [12]

The original perceptron can be considered an ANN composed by one artificial neuron. It is, then, very similar to the concept of artificial neuron.

> **Definition:** *The artificial neuron is a mathematical function conceived as a model of biological neurons.*

Let's see now a Perceptron scheme:

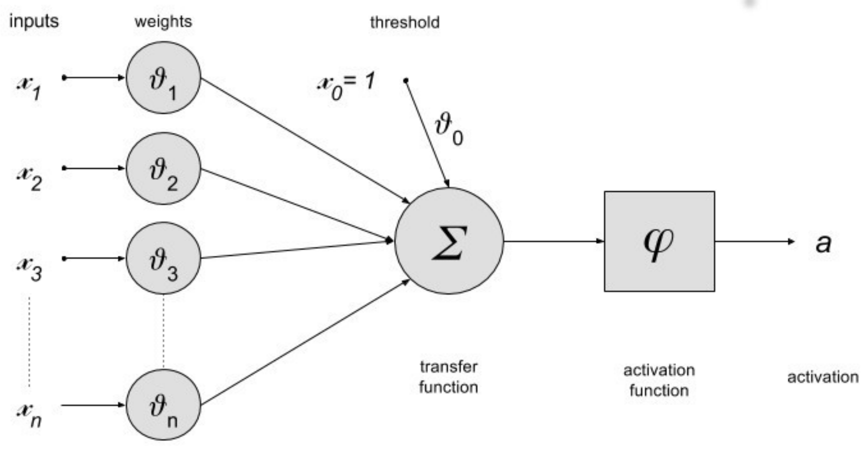


Figure 2.3: Single layer Perceptron scheme [9]

A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and 0 otherwise. More precisely, given inputs x_1 through x_n, the output $a(x_1, ..., x_n)$ computed by the perceptron is:

$$a(x_1, ..., x_n) = \begin{cases} 1 & \text{if} \quad \theta_0 + \theta_1 x_1 + \theta_1 x_2 + ... + \theta_n x_n > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

where each θ_i is a real-valued constant, or weight, that determines the contribution of input x_i to the perceptron output. Notice the quantity $(-\theta_0)$ is a threshold that the weighted combination of inputs $\theta_1 x_1 + \theta_1 x_2 + \cdots + \theta_n x_n$ must surpass in order for the perceptron to output a.To simplify notation, we imagine an additional constant input $x_0 = 1$, allowing us to write the above inequality as $\sum_{i=0}^{n} \theta_i x_i > 0$, or in vector form as $\boldsymbol{\theta}^T \cdot \boldsymbol{x} > 0$. The original activation function φ maps the output to 1 or 0. This function is the *step function*:

$$\varphi(\boldsymbol{x}) = step(\boldsymbol{\theta}^T \cdot \boldsymbol{x})$$

where

$$sgn(z) = \begin{cases} 1 & \text{if} \quad z > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

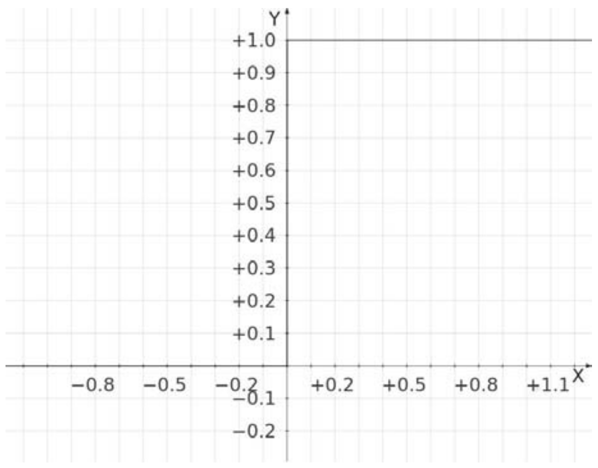


Figure 2.4: Step function.

Learning a perceptron involves choosing values for the weights $\theta_0, \cdots, \theta_n$. Therefore, the space H of candidate hypotheses considered in perceptron learning is the set of all possible real-valued weight vectors.

$$H = \{\boldsymbol{\theta} \mid \boldsymbol{\theta} \in \mathbb{R}^{\{n+1\}}\}$$

2.3.3 Deep Neural Networks

The concepts behind the preceptron are the ones of modern neural network and then of Deep Learning. An evolution of the perceptron is a multilayer network with *hidden* layers between the input and the output.

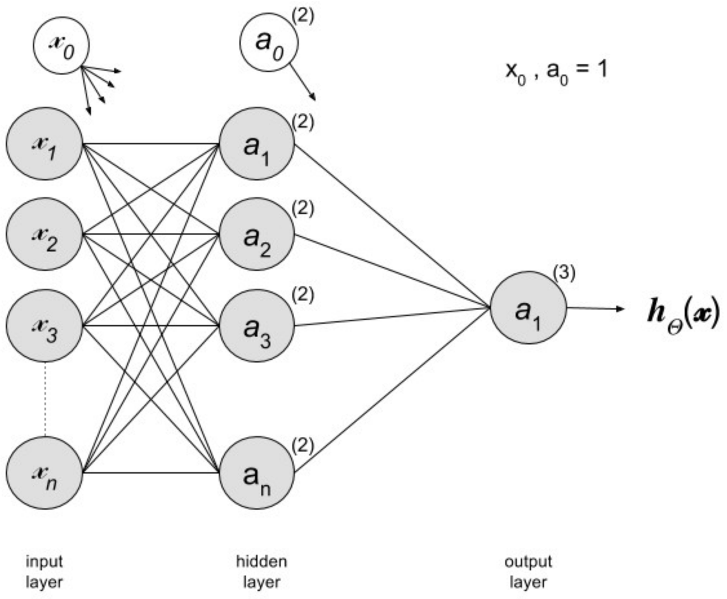


Figure 2.5: 3-layer fully connected neural network.

In figure 2.5 a 3-layer fully connected network is shown. The first layer is composed by the inputs values. For each activation $a_i^{(2)}$ in the second layer a independent vector of weight $\boldsymbol{\theta}_i^{(1)}$ is used. Therefore, is possible to write:

$$a_i^{(2)} = \boldsymbol{\theta}_i^{(1)} \cdot \boldsymbol{x}$$

where $\boldsymbol{\theta}_i^{(1)}$ is the row i of the matrix $\boldsymbol{\theta}^{(1)}$ that maps layer 1 to layer 2. The activation function is used for each output giving

$$a_i^{(2)} = \varphi(\boldsymbol{\theta}_{i0}^{(1)} x_0, \boldsymbol{\theta}_{i1}^{(1)} x_1, \boldsymbol{\theta}_{i2}^{(1)} x_2, \boldsymbol{\theta}_{i3}^{(1)} x_3, \boldsymbol{\theta}_{i4}^{(1)} x_4)$$

or, in a compact way:

$$\boldsymbol{a}^{(2)} = \varphi(\boldsymbol{\theta}^{(1)} \cdot \boldsymbol{x})$$

Each activation $a_i^{(2)}$ is then mapped to $a_1^{(3)}$ through a second weight matrix $\boldsymbol{\theta}^{(2)}$. Furthermore is possible map the input $\boldsymbol{x}$ directly to the output $a_1^{(3)}$ using the notation $a_1^{(3)} = h_\Theta(\boldsymbol{x})$ where $\Theta = \{\boldsymbol{\theta}^{(1)}, \boldsymbol{\theta}^{(2)}\}$.

2.3.4 Activation function

The activation function is very important for deep networks. The step or sign functions provide a very strong decision that is not very good with deep networks. Before explore better what a deep network is, let's see a soft activation function:

Logistic unit

This function allows to propagate more of the initial information and this is useful in the case of deep networks.

$$sig(z) = \frac{1}{1 + exp(-z)}$$

$$\frac{d(sig(z))}{dz} = \frac{e^z}{(e^z + 1)^2}$$

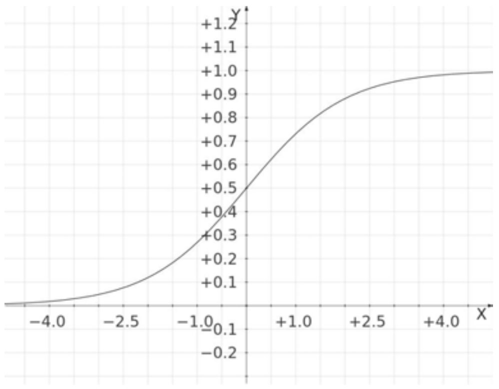


Figure 2.6: Logistic function.

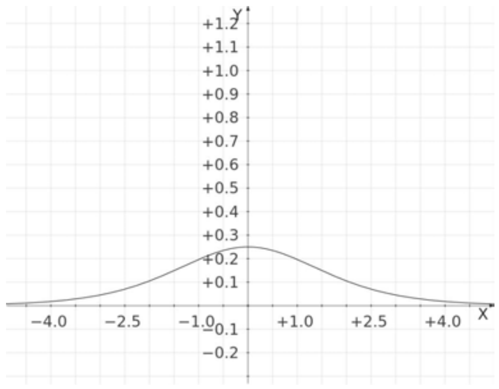


Figure 2.7: First derivative.

Rectified linear unit

The Rectified Linear Unit is an activation function useful for deep linear network as will be possible to see in the next sections.

$$\text{ReLU} : f(x) = \max(0, x) \tag{2.3}$$

$$\frac{df(z)}{dz} = step(z)$$

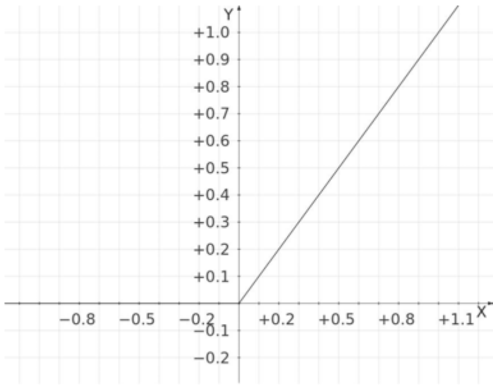


Figure 2.8: ReLu function

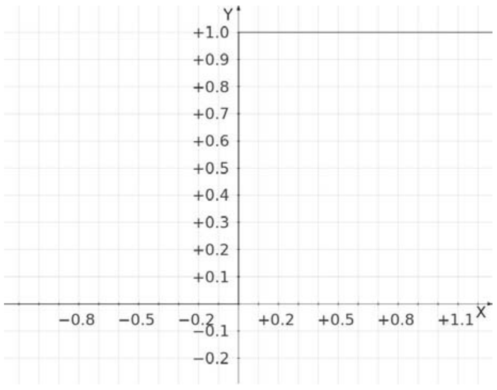


Figure 2.9: ReLU First derivative.

2.4 Backpropagation algorithm

Until now, only the structure of the network has been discussed. The network is, from a computational point of view, a chain of parallel and serial calculations which map an input to an output. But how is possible to teach something to this architecture? How is possible that this system can learn something? To answer this question is necessary to introduce the backpropagation algorithm. This method allows the network to modify the way how the steps are computed making possible to adjust the output. Recalling the general structure of supervised learning, this is done through the training data in the training step: knowing the output related to each input is possible to evaluate the error with the respect to the output of the network. The backpropagation step is the algorithm that allows to change the network trying to reduce the computed error. Let's see now the backpropagation algorithm in detail.

2.4.1 Training set

The training set is composed of m inputs $\boldsymbol{x}^{(i)}$ with the respective outputs, or *labels*, $\boldsymbol{y}^{(i)}$, where $i = 1, \cdots, m$ is the index of each sample.

$$X = \underbrace{\text{ffl}} \underbrace{\begin{bmatrix} \text{———} \ \boldsymbol{x}^{(1)} \ \text{———} \\ \text{———} \ \boldsymbol{x}^{(2)} \ \text{———} \\ \text{———} \ \boldsymbol{x}^{(3)} \ \text{———} \\ \vdots \\ \text{———} \ \boldsymbol{x}^{(m)} \ \text{———} \end{bmatrix}}_{\text{n columns}} \qquad Y = \underbrace{\begin{bmatrix} \text{——} \ \boldsymbol{y}^{(1)} \ \text{——} \\ \text{——} \ \boldsymbol{y}^{(2)} \ \text{——} \\ \text{——} \ \boldsymbol{y}^{(3)} \ \text{——} \\ \vdots \\ \text{——} \ \boldsymbol{y}^{(m)} \ \text{——} \end{bmatrix}}_{\text{k columns}} \tag{2.4}$$

In the representation (2.4) X is the matrix where each is row is a vector representing an input. The single input is not always a linear vector but the generality is not lost because nothing keep from reshape each row to a multi dimensional vector as an image or a $3D$ data. Y is the respective *labels* matrix: for each row a label vector is enumerated. With reference to the previous neural network architecture is possible to write:

$$X = \left[\begin{array}{cccc} \boldsymbol{x}_1 & \boldsymbol{x}_2 & \cdots & \boldsymbol{x}_n \\ \hline x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ x_1^{(3)} & x_2^{(3)} & \cdots & x_n^{(3)} \\ & & \vdots & \\ x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{array}\right] \qquad Y = \left[\begin{array}{cccc} \boldsymbol{y}_1 & \boldsymbol{y}_2 & \cdots & \boldsymbol{y}_n \\ \hline y_1^{(1)} & y_2^{(1)} & \cdots & y_n^{(1)} \\ y_1^{(2)} & y_2^{(2)} & \cdots & y_n^{(2)} \\ y_1^{(3)} & y_2^{(3)} & \cdots & y_n^{(3)} \\ & & \vdots & \\ y_1^{(m)} & y_2^{(m)} & \cdots & y_n^{(m)} \end{array}\right] \tag{2.5}$$

In the (2.5) first row of the first matrix represents the vector of all the input for each position of the first layer. In this case the network must have n inputs. The symbol $x_j^{(i)}$ with $i = 1, \cdots, m$ and $j = 1, \cdots, n$ represents the element that feed the j^{th} input of the network for the i^{th} sample. The same is valid for the outputs $y_l^{(i)}$ with $l = 1, \cdots, k$. Must be noticed that, with reference to the architecture of figure 2.5, $k = 1$, and $\boldsymbol{y}^{(i)}$ becomes a single element $y^{(i)}$.

2.4.2 Loss function

In order to see how to reduce the output error of a neural network is necessary to define what this error is. Therefore, a loss function is now defined.

$$\mathcal{L}(\Theta) = \frac{1}{m}\sum_{i=1}^{m} E^{(i)} \tag{2.6}$$

where m is the number of training samples in the training set and $E^{(i)}$ is the error between the output of the network and the expected error for sample i.

$$E^{(i)} = E^{(i)}(h_\Theta(\boldsymbol{x})) = \frac{1}{2}||\boldsymbol{y}^{(i)} - \underbrace{h_\Theta(\boldsymbol{x}^{(i)})}_{=\boldsymbol{a}^{(L)}}||^2 = \frac{1}{2}\sum_k (y_k^{(i)} - a_k^{(L)})^2 \\ \boldsymbol{x}, \boldsymbol{y} \quad \text{fixed} \tag{2.7}$$

where $\boldsymbol{a}^{(L)}$ is the activation vector on the last layer of the network and $\boldsymbol{y}^{(i)}$ the label of the i^{th} sample. E varies with Θ, and this is the what to optimise to reduce E.

2.4.3 Gradient Descent Rule

The main idea is to reduce Θ with the following rule:

$$\Theta \to \Theta - \eta\frac{\delta E}{\delta\Theta} \tag{2.8}$$

where η is called *learning rate* and $\frac{\delta E}{\delta \Theta}$ is the rate of variation of error E with the respect to Θ. Recalling that theta is the cascade of matrices that map the input to the output the values to find are related to $\Theta_{ij}^{(l)}$, the weight in position ij of the matrix $\Theta^{(l)}$ that map the layer l to the layer $l+1$, $l = 1, \cdots, L$.

2.4.4 Local Error

Now will be explored how the local weights are updated. Let's be $z_i^{(l)} = \boldsymbol{\theta}_{i0}^{(1)}x_0, \boldsymbol{\theta}_{i1}^{(1)}x_1, \boldsymbol{\theta}_{i2}^{(1)}x_2, \cdots, \boldsymbol{\theta}_{in}^{(1)}x_n$ the weighted input of layer l of neuron i. The update step performs:

$$z_i^{(l)} \to z_i^{(l)} + \Delta z_i^{(l)} \tag{2.9}$$

and the error changes with

$$E \to E + \delta_i^{(i)} \cdot \Delta z_i^{(l)} \quad \text{with} \quad \delta_i^{(i)} = \frac{\delta E}{\delta z_i^{(l)}} \tag{2.10}$$

Therefore, the value $\delta_i^{(i)}$, error at i^{th} neuron of layer l, is fundamental for the update:

$$\begin{aligned}
&\text{if} \quad \left|\frac{\delta E}{\delta z_i^{(l)}}\right| \gg 0 \quad \Rightarrow \quad \text{improvement is possible} \\
&\text{if} \quad \left|\frac{\delta E}{\delta z_i^{(l)}}\right| \simeq 0 \quad \Rightarrow \quad \text{improvement is near 0, optimal node}
\end{aligned} \tag{2.11}$$

Output error

The *output error*, at layer $l = L$ is:

$$\delta_i^{(L)} = \frac{\delta E}{\delta a_i^{(l)}} \cdot \frac{\delta a_i^{(l)}}{\delta z_i^{(l)}} = (a_i^{(L)} - \boldsymbol{y}_i) \cdot \varphi'(z_i^{(L)}) \tag{2.12}$$

recalling that:

$$a_i^{(l)} = \varphi(z_i^{(l)})$$

$$\Rightarrow \quad \frac{\delta a_i^{(L)}}{\delta z_i^{(L)}} = \frac{\delta \varphi(z_i^{(L)})}{\delta(z_i^{(L)})} = \varphi'(z_i^{(L)})$$

and that:

$$E = \frac{1}{2}\sum_k (\boldsymbol{y}_k - a_k^{(L)})^2$$

$$\Rightarrow \frac{\delta E}{\delta a_i^{(L)}} = \frac{1}{2} \cdot 2(a_i^{(L)} - \boldsymbol{y}_i)$$

The 2.12 is a general formula that does not depend on the φ activation function. In case φ is the *sigmoid* the 2.12 becomes:

$$\delta_i^{(L)} = (a_i^{(L)} - \boldsymbol{y}_i) \cdot a_i^{(L)}(1 - a_i^{(L)}) \tag{2.13}$$

Considering all the inputs for each sample is possible to write the 2.12 in vectorial form:

$$\begin{gathered}
\boldsymbol{\delta}^{(L)} = \nabla_a E \odot \varphi'(\boldsymbol{z}^{(L)}) \\
\text{with} \quad \odot \quad \text{componentwise multiplication}
\end{gathered} \tag{2.14}$$

Error at layer l

The error at layer l depends on the next layer $l+1$:

$$\begin{aligned} \delta_i^{(l)} = \frac{\delta E}{\delta z_i^{(l)}} = \sum_j \frac{\delta E}{\delta z_j^{(l+1)}} \cdot \frac{\delta z_j^{(l+1)}}{\delta z_i^{(l)}} \\ \delta_i^{(l+1)} = \frac{\delta E}{\delta z_i^{(l+1)}} \end{aligned} \tag{2.15}$$

with $j = 1, \cdots, v$ element in the layer $l+1$. Knowing that:

$$z_j^{(l+1)} = \sum_k \Theta_{jk}^{(l)} \cdot a_k^{(l)} = \sum_k \Theta_{jk}^{(l)} \cdot \varphi(z_k^{(l)}) \tag{2.16}$$

$$\Rightarrow \qquad \frac{\delta z_j^{(l+1)}}{\delta z_i^{(l)}} = \Theta_{ji}^{(l)} \varphi'(z_i^{(l)}) \tag{2.17}$$

where the 2.17 is computed with the *Jacobian of a composite function*, recalled in Appendix A. The equation 2.15 becomes:

$$\delta_i^{(l)} = \sum_j \delta_j^{(l+1)} \cdot \frac{\delta z_j^{(l+1)}}{\delta z_i^{(l)}} = \sum_j \Theta_{ji}^{(l)} \delta_j^{(l+1)} \varphi'(z_i^{(l)}) \tag{2.18}$$

and considering the entire layers, the vectorial form is:

$$\delta_i^{(l)} = \left[(\boldsymbol{\Theta}^{(l)})^T \cdot \boldsymbol{\delta}^{(l+1)} \right] \odot \varphi'(\boldsymbol{z}^{(l)}) \tag{2.19}$$

Layer error update

It is finally possible to write the variation of the error with the respect to $\Theta^{(l)}$, matrix of weight that maps layer l to layer $l+1$ as a function of the input $\boldsymbol{a}^{(l)}$ and of $\boldsymbol{\delta}^{(l+1)}$.

$$\frac{\delta E}{\delta \Theta_{ij}^{(l)}} = \frac{\delta E}{\delta z_i^{(l+1)}} \cdot \frac{\delta z_i^{(l+1)}}{\delta \Theta_{ij}^{(l)}} = \delta_i^{(l+1)} \cdot a_j^{(l)} \tag{2.20}$$

where $\delta_i^{(l+1)}$ is computed through the *backward* step and $a_j^{(l)}$ is previously computed through the *forward* step. Considering all the elements i and j in a vector the 2.20 becomes:

$$\frac{\delta E}{\delta \Theta^{(l)}} = \boldsymbol{\delta}^{(l+1)} \cdot \left(\boldsymbol{a}^{(l)}\right)^T \tag{2.21}$$

Recalling the 2.8, the weights are then updated with:

$$\Theta^{(l)} \quad \rightarrow \quad \Theta^{(l)} - \eta \cdot \boldsymbol{\delta}^{(l+1)} \cdot \left(\boldsymbol{a}^{(l)}\right)^T \tag{2.22}$$

2.4.5 Backpropagation steps

With the achievements of the last sections, the backpropagation algorithm can be summarised in few steps. Given a multilayer neural network with activation φ the bacpropagation algorithm requires:

- The *Training set*;
- the learning rate η;
- the optimization function that defines the error E;
- a *termination condition*, which can be a maximum number of steps or a minimum error reduction rate.

Once the initial requirements are satisfied the algorithm can be implemented:

Algorithm 1 Backpropagation

```
1: Initialize all network weights to small random numbers (e.g., between
   $-.05$ and $.05$).
2: while Termination condition is False do
3:     for each $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ in training set do
       ▷ Propagate the input forward through the network
4:         Input the instance $\boldsymbol{x}^{(i)}$ to the network and compute the output
           $\boldsymbol{a}^{(l)}$ of each node of the network. .
           ▷ Propagate the errors backward through the network
5:         Compute the error $\boldsymbol{\delta}^{(L)} = \nabla_a E \odot \varphi'(\boldsymbol{z}^{(L)})$ of the network output
6:         for $l$ in range $\{L \cdots 1\}$ do
7:             Compute the error for each of the $l$ hidden layers:
               $\delta_i^{(l)} = \left[ (\boldsymbol{\Theta}^{(l)})^T \cdot \boldsymbol{\delta}^{(l+1)} \right] \odot \varphi'(\boldsymbol{z}^{(l)})$.
8:         end for
9:         for $l$ in range $\{L \cdots 1\}$ do
10:            Compute the variation of the error with the respect
               to the weights for each of the $l$ hidden layers:
               $\frac{\delta E}{\delta \Theta^{(l)}} = \boldsymbol{\delta}^{(l+1)} \cdot \left(\boldsymbol{a}^{(l)}\right)^T$.
11:            Update the layers following the Stochastic Gradient Descent:
               $\Theta^{(l)} \quad \rightarrow \quad \Theta^{(l)} - \eta \cdot \boldsymbol{\delta}^{(l+1)} \cdot \left(\boldsymbol{a}^{(l)}\right)^T$ .
12:        end for
13:    end for
14: end while
```

2.5 Convolutional Neural Networks

The networks seen until now are composed of neurons completely connected to each other. This means a huge number of weights to be memorised with the increase of the layers and neurons per layer. The problem becomes important from a memory requirement and training time point of view. Indeed in modern neural networks the number of parameters can be of several millions, and the convergence becomes very slow. In the field of images the data has can be exploited in a better way and a fully connected architecture is not necessary neither efficient. In fact images have usually very high correlation between near pixels and low correlation between far points. This can be used designing a more sparse neural architecture highly connected only *locally*.

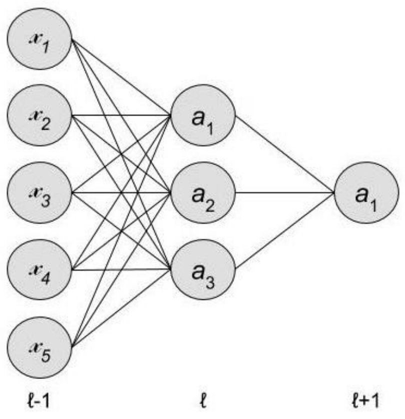


Figure 2.10: Fully connected network

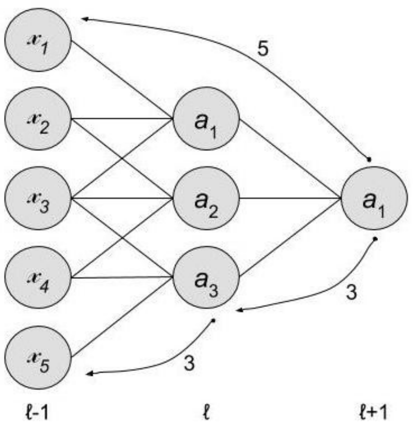


Figure 2.11: Sparse network

In figure 4.9 each neuro is connected to 3 adjacent neurons of the previous layer. In the case in figure the receptive field of each layer to the previous is $RF = 3$, equal to the number of neurons of the layer $l - 1$ connected to a neuron of layer l. The RF of the output with the respect to the input is 5; all the input layers are still seen by the output neuron.

2.5.1 Convolutional layer

The convolutional layer is design mainly for image data. Therefore, is reasonable to choose an image as input for the network. The structure of a coloured image $\boldsymbol{x}$ is:

$$\boldsymbol{x} \in \mathbb{R}^{\{n_1 \times n_2 \times n_3\}}, \quad \text{with} \quad n_1 = 3$$

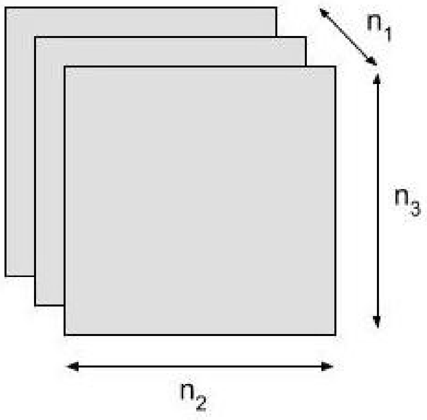


Figure 2.12: RGB image data structure.

The matrices $\Theta_i = k_i$, $i = 1, \cdots, m$ matrix of weights that maps the input to the output of the layer, is of dimension :

$$k_i \in \mathbb{R}^{\{m \times n_1 \times p_1 \times p_2\}}, \quad \text{with} \quad n_1 = 3, \; p_1 < n_2, \; p_2 < n_3$$

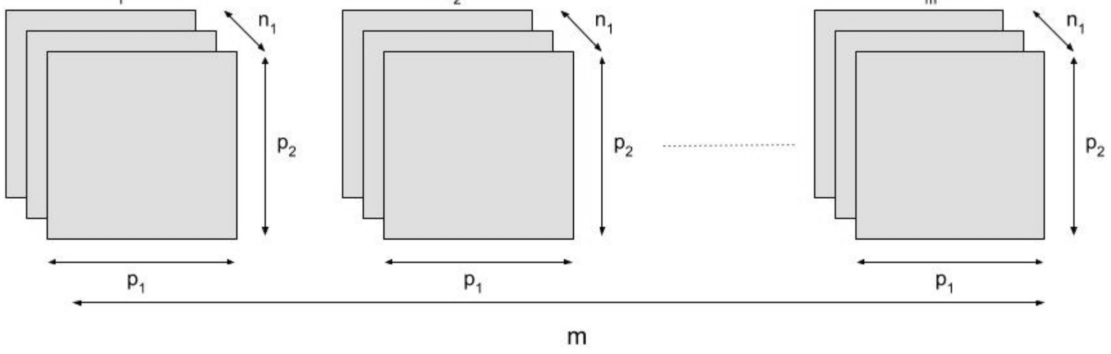


Figure 2.13: Weights matrix structure.

The outputs y_i are computed with a convolution between the input image $\boldsymbol{x}$ and the the k_i. The k_i can, then, be considered as *filter kernels.*

$$y_i = \boldsymbol{x} * k_i + b_i \quad i = 1, \cdots, m \tag{2.23}$$

where b_i is the *bias* term. k_i and b_i are the terms learned during the training phase.

2.5.2 Pooling layer

The pooling layer is fundamental to exploit high level path from the input and to reduce the number of parameters. The function implemented by this layer is:

$$\begin{aligned} \|\boldsymbol{x}\|_p &= \left(\sum_i |x_i|^p\right)^{\frac{1}{p}} \\ \|\boldsymbol{x}\|_p &= \max(\boldsymbol{x}) \quad p \to +\infty \end{aligned} \tag{2.24}$$

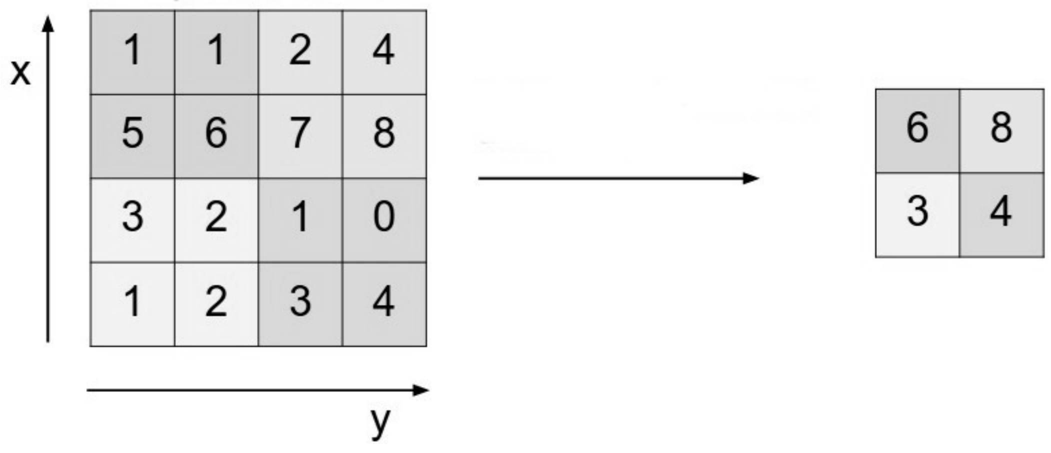


Figure 2.14: Maxpooling layer[13] of size 2×2

2.6 Classifier performances

2.7 Common problems

Overfitting
imbalaced classes

Chapter 3

Literature review: Deep Learning for retinal images analysis

Literature review: Deep Learning for retinal images analysis

	TITLE	AUTHORS	PUBLISHED IN	DATA TYPE	METHODS	DATA	RESULTS	Acronyms
1	Automated retinopathy of prematurity case detection with convolutional neural network	Daniel E. Worral, Clare M.Wilson	Deep Learning and Data Labeling for Medical Applications: First International Workshop, LABELS 2016, and Second International Workshop, DLMIA 2016, Held in Conjunction with MICCAI 2016, Athens, Greece, October 21, 2016, Proceedings	Fundus images	**1) Preprocessing and Data Augmentation** **Centering and cropping to 240 x 240 px** **High pass filter on RGB channel** **Data Augm. With flips, rotation and subcrop.** **2)Classifier (per image or per exam)** **2-way softmax classifier stacked on top of a pretrained GoogLeNet (on ImageNet data)** **Bayesan CNN** **3) Visualization** **Visualization of deseased regions by CNN feature maps**	**Canada dataset:** 1459 images from 35 patients (640 x 480 px) Train and Test **London dataset** 106 individually labeled images Only Test	Per image **Raw accuracy: 0.918** **Sensitivity: 0.825** **Specificity: 0.983** Per exam **Raw accuracy: 0.936** **Sensitivity: 0.954** **Specificity: 0.947**	CNN = Convolutional Neural Network ROP = Retinopathy of Prematurity
2	A deep learning method for microaneurysm detection in fundus images	Juan Shan Lin Li	2016 IEEE First International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)	Fundus images	**1) Preprocessing** **Green channel used.** **Intensity rescaled on a [0, 1] scale.** **2) Feature extraction** **Network architecture made by a SSAE (2 Sparse Autoencoder layers in this case) as first stage and by SMC as second stage.** **Training iteration for SSAE = 100.** **Training iteration for SMC = 50.** **Fine-tuning with training iteration = 5.**	**89 images form DIARETDB:** **-> 2182 MA patches** **-> 6230 healthy patches** (25x25 px areas choosed randomly more than 25px away from a MA annotated patches)	**Performances evaluated on a 10-fold cross validation:** **Precision: 91.57%** **Specificity: 91.6%** **Accuracy: 91.38%** **AUC: 96.2%**	CNN = Convolutional Neural Network DR = Diabetic retinopathy MA = microaneurysm SSAE = Stacked Sparse Autoencoder SMC = Softmax Classifier
3	Convolutional Neural Networks for Diabetic Retinopathy	Harry Pratta, Frans Coenenb, Deborah M Broadbentc, Simon P Hardinga, Yalin Zhenga	International Conference On Medical Imaging Understanding and Analysis 2016, MIUA 2016,6-8 July 2016, Loughborough, UK	Fundus images	**1) Preprocessing** **Colour normalization with OpenCV (Kaggle database is very heterogeneous)** **Images are resized to 512x512 px size in order to reduce complexity.** **2) Network (Keras on Theano)** **10 layers CNN from 32 to 512 filters layer.** **2 fully connected layers** **1 fully connected layer with softmax** **3) Training** **Pre-training on 10290 images with 120 epochs.** **Training on the remainder 78000 images for a further 20 epochs.** **To reduce overfitting back-propagation weights updates are weighted on the number of DR classified images in the current batch of training.** **At each epoch data augmentation is used (rotations flips shifts for each image).** **Stochastic gradient descent with Nestrov momentum is used.** **Learning rate of 0.0001 for first 5 epochs ond then 0.0003 -> 350 hours** **4) Test** **5000 images saved for validation**	**80000 images from Kaggle Dataset:** **512x512 input resolution**	**5 label classification:** **-No DR** **-Mild DR** **-Moderate DR** **-Severe DR** **-Proliferative DR** Specificity: 95% Accuracy: 75% Sesitivity: 30% **more than 350 hours of training time on NVIDIA K40c**	CNN = Convolutional Neural Network DR = Diabetic retinopathy
4	Deep Retinal Image Understanding	Kevis-Kokitsi Maninis, Jordi Pont-Tuset, Pablo Arbelaez, and Luc Van Gool	Medical Image Computing and Computer-Assisted Intervention – MICCAI 2016 19th International Conference, Athens, Greece, October 17-21, 2016, Proceedings, Part II	Fundus images	**1) Network design** Start from VGG network. Remove the fully connected layers at the end: only CNN architecture is keeped. Deepers layers are pretrained with high number of general images. Between the pooling layers feature maps are extracted. Each feature map is combined by a last convolutional layer Feature maps combination are divided between the 4 more dense ones and the 4 more coarser. **2) OD and Vessels** The network is used both for vessel and OD segmentation The coarser (deeper) map are not very useful for vessels, but is good for OD. The first feature map volume is used for vessels (the denser one), and the second for OD segmentation. **3) Training** 20000 iterations. Stochastic descent with momentum. Learning rate = 10-8 gradually decreased as training proceeds. (avoid overfitting) All RGB channel are used. Classic data augmentation (rotation, shifts, flips) is used. (avoid overfitting) **4) Testing** All images are segmented manually by two doctors and by the machine. The first doctor segmentation is used as gold standard. The second doctor and the machine segemntation are compared to the gold standard in a human-machine performances comparison.	Vessel segmentation: **40 images from DRIVE** **20 images from STARE** Optic Disc segmentation: **110 images from DRIONS-DB** (60 for training, 50 for test) **159 images from RIM-ONE** (99 for training, 60 for test) All images are labeled for segmentation	**Test and training executed on TITAN X (Maxwell) GPU** **Vessels:** **85ms per image on DRIVE test** **104ms per image on STARE test** **OD:** **65ms per image on DRIONS-DB test** **104ms per image on RIM-ONE test** **The machine result are comparable to the second doctor segmentation when compared to the gold standard (the first doctor work).**	OD = optic disc

5	USING DEEP LEARNING FOR ROBUSTNESS TO PARAPAPILLARY ATROPHY IN OPTIC DISC SEGMENTATION	Ruchir Srivastava, Jun Cheng	2015 IEEE 12th International Symposium on Biomedical Imaging (ISBI)	Fundus images	**1) Preprocessing** **Use of red channel in images not oversaturated. Combination of green and blue channels in the other ones.** **Region of interest of 800x800 pixels cropped.** **Resize to 300x300 for faster processing.** **2) Feature extraction** **Pixelwise classification with patchwise training.** **DeepLearnToolbox used for DNN implementation** **Pixel with decision value > 0.7 classified as OD.**	**230 images from Singapore Malay Eye Study (all with PPA and OD manually labeled)** **-> 2048 x 3072 (cropped 800x800 area resized to 300x300 px used)** **->200000 patches from all the images used as trainingset (half from OD region, half from non OD region)**	**Result given as the mean overlapping error μe between the result and the ground truth.** **μe= 9.7%**	OD = optic disk PPA = parapapillary atrophy
6	Detection of Exudates in Fundus Photographs using Convolutional Neural Networks	Pavle Prentasic , Sven Loncaric´	2015 9th International Symposium on Image and Signal Processing and Analysis (ISPA)	Fundus images	**1) Preprocessing and channel selction** **Green channel is selected.** **Patches of 65x65 extracted from each pixel.** **The database is splitted disjointly in training and test set.** **2) Classification and OD detection** **A 4 layers CNN with final fully connected layer is trained. All the pixel belonging to an exudate area are used for training. In ordre to have a balanced training set the number of patches is doubled with random pixels from healthy labeled areas.** **From each image the OD is detected as the largest object with high intensity pixel inside image given a choosen threshold.** **3) OD Masking** **The OD area is masked** **4) Testing** **For each imaged the number of TP, FP and FN is calculated using a threshold approach.**	**50 colour images from DRiDB with ground truth avilable** **65x65 pixelwise patches**	**= 0.77** **= 0.77** **= 0.77** **10 hours of training in NVIDIA TESLA K20c**	OD = optic disk TP = true positive FP = false positive FN = false negative S = Sensitivity PPV = positive predictive value

Chapter 4

Methodology

The work done in this project has been implemented in Keras([14][15]), a neural network API for Tensorflow([16][17][18]) and in Torch([19][20]). Tensorflow is a library written in python [21] and c++; Torch is an interpreter for LuaJIT ([22], [23]). All the libraries manage properly Nvidia CUDA ([24]) for GPU acceleration. The dataset used is the one provided for the Diabetic Retinopathy Detection challenge on Kaggle ([25]) by EyePACS ([26]).

4.1 Hardware setup

Deep Learning training tasks are very heavy from a computational point of view. Luckily, as already seen, forward and backward steps are largely parallelizable. This is done using GPU, instead of CPU, and exploiting their highly multi-core architecture. In particular the core of Tensorflow and Torch can handle CUDA that allows to use the GPU to run *c++* code. The machine comprises 1x 250GB OS 'disk' (NVMe 32Gbps) hosting Ubuntu 16.04 LTS, 64GB RAM memory, 1x i7-6700 CPU, 1x 4TB data disk and 1x 4TB backup disk (both spinning) across SATA3 @ 6Gbps.It two GPUs. Both are reference Nvidia-supplied Titan X (Pascal architecture), 12GB VRAM and 3584 CUDA cores each, residing in PCIe 3.0 x16 slots on an Asus Z170 Pro motherboard with standard CPU/GPU clock rates. Each GPU is used for a single task per time and each task run on only one GPU.

4.2 Software setup

As anticipated before Tensorflow, Keras and Torch are installed on the machine. The machine is used as a server in order to simplify access to the GPUs for multiple users. Therefore the code runs in remote and a special environment setup is necessary to have full control on the code during running and debug sessions.

4.2.1 Python

Python is the lenguage used by Keras and Tensorflow. The IDE used to have the best control possible on code development is JetBrains PyCharm

Professional ([27]). The setup for a remote interpreter has been used to produce a guide with all the details of the setup and the environment used.

Titan1 guide for DeepLearning applications

Author: Enrico Vincenzi
Last update: 25/11/2016

This guide is thought for an easy access to Titan1 server for deep learning. PyCharm is here used as standard python IDE because is simple, light and free for University members. The guide is divided in two main parts:

- PyCharm setup and configuration for Titan1 server
- SSH session section for time consuming runs

SSH sessions are launched inside PyCharm in order to have a better integration of the development environment, but everything shown under SSH section is valid also outside PyCharm.

PyCharm

Requirements

- **PyCharm professional (version > 2016.3)**

  https://www.jetbrains.com/pycharm/download;

  Download professional version;

  PyCharm professional version is NOT a free software but is free for students and teachers. To activate 1-year free student licence you need to register here (Apply now section) with a University email account.

- **Titan1 server account**

  Server IP: 134.36.37.238

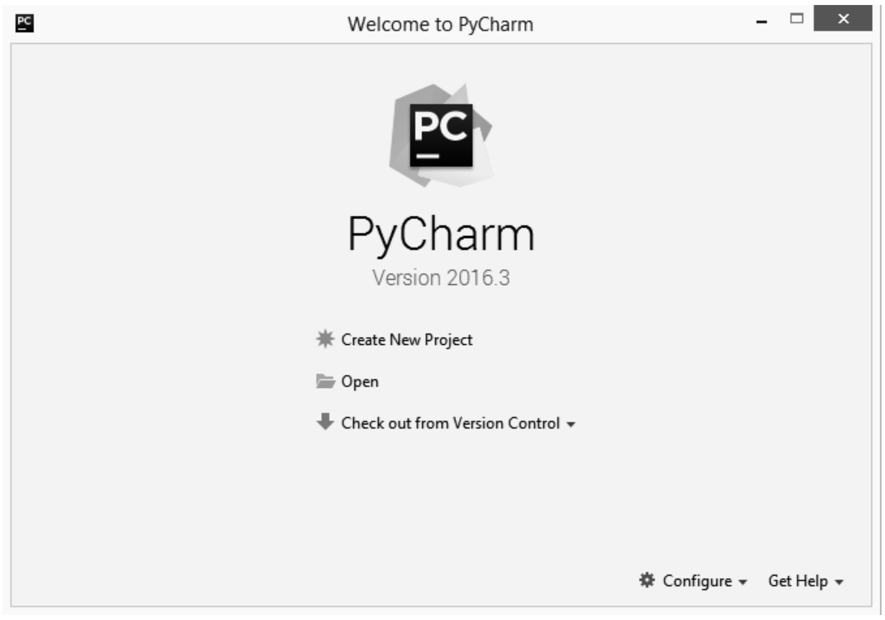
  Ask Derek for credentials.

Install PyCharm

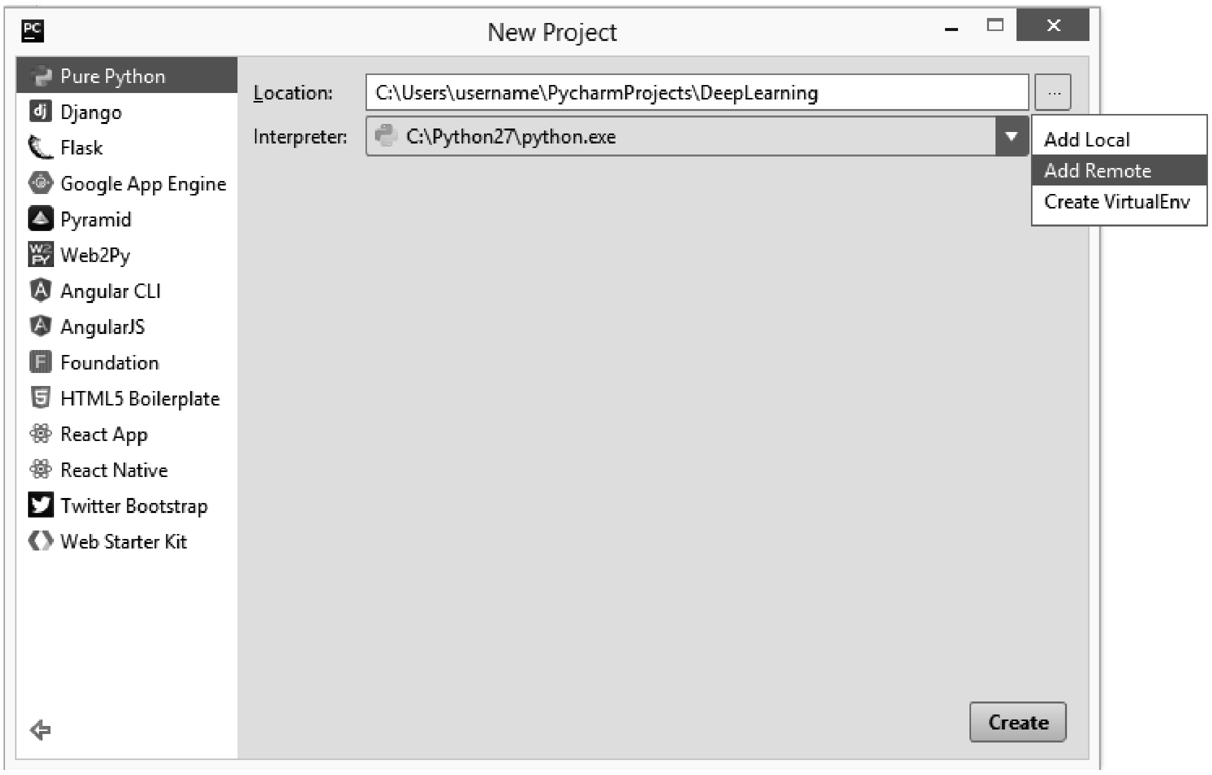
Install PyCharm professional on your computer. Access with your full or University account.

Link PyCharm project to a remote Interpreter

After the initial setup you should see a window like the next one:



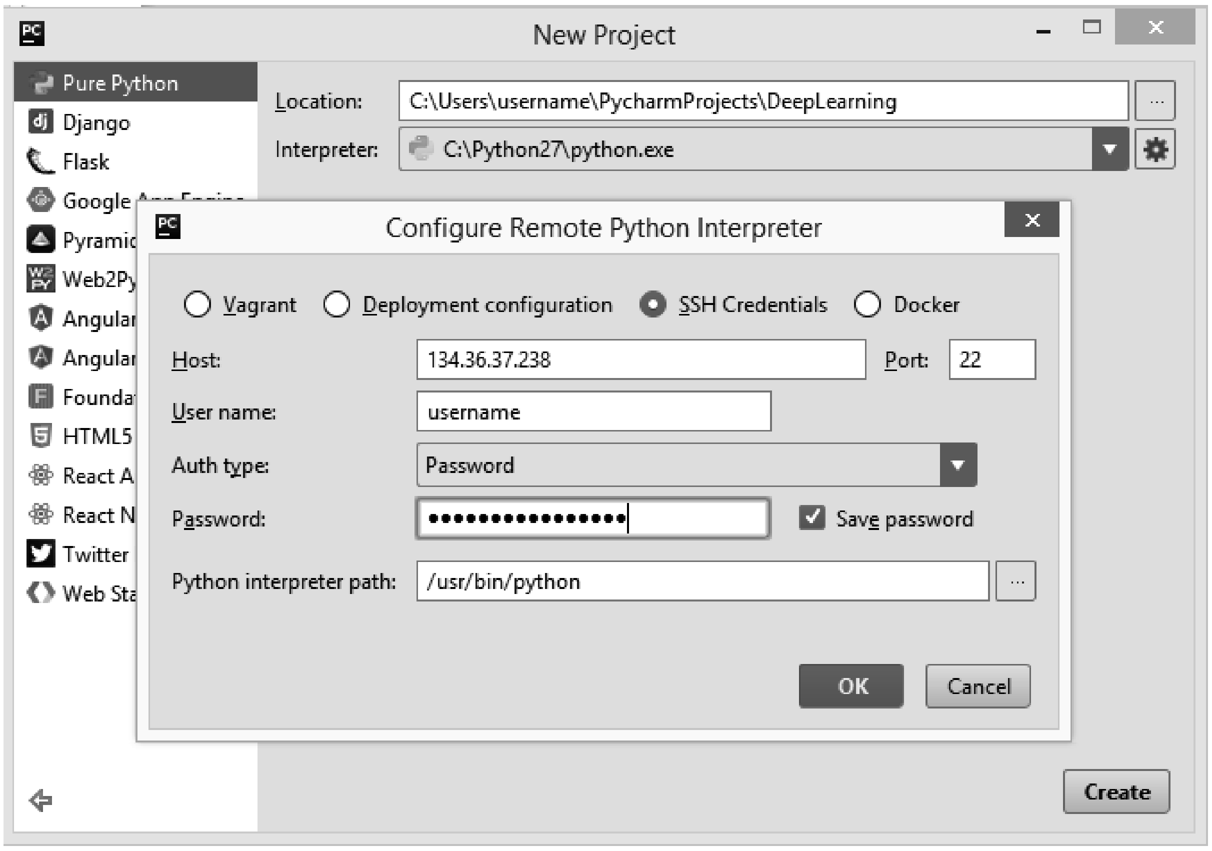
Click on Create New Project


New Project

Location: C:\Users\username\PycharmProjects\DeepLearning

Interpreter: C:\Python27\python.exe

Press on the *gear* near the interpreter row and select *add remote*.


New Project

Location: C:\Users\username\PycharmProjects\DeepLearning

Interpreter: C:\Python27\python.exe

Configure Remote Python Interpreter

( ) Vagrant ( ) Deployment configuration [✓ SSH Credentials] ( ) Docker

Host: 134.36.37.238 Port: 22

User name: username

Auth type: Password

Password: ••••••••••••••••• [✓] Save password

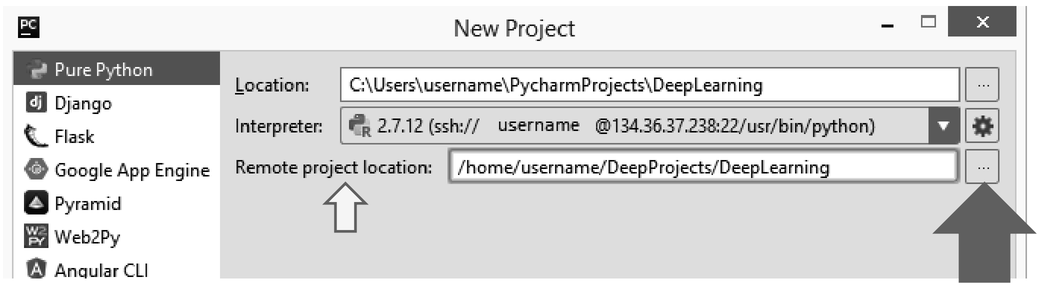
Python interpreter path: /usr/bin/python

Select SSH Credentials

- Host: 134.36.37.238
- Fill with your credentials.

IMPORTANT: Set your *'remote project location'* (is possible to create a new folder with right click in the window opened clicking on ... button); The default setting generates a project space outside your account.

If everything worked set the remote project location (where to upload the code):



New Project

Location: C:\Users\username\PycharmProjects\DeepLearning

Interpreter: 2.7.12 (ssh:// username @134.36.37.238:22/usr/bin/python)

Remote project location: /home/username/DeepProjects/DeepLearning

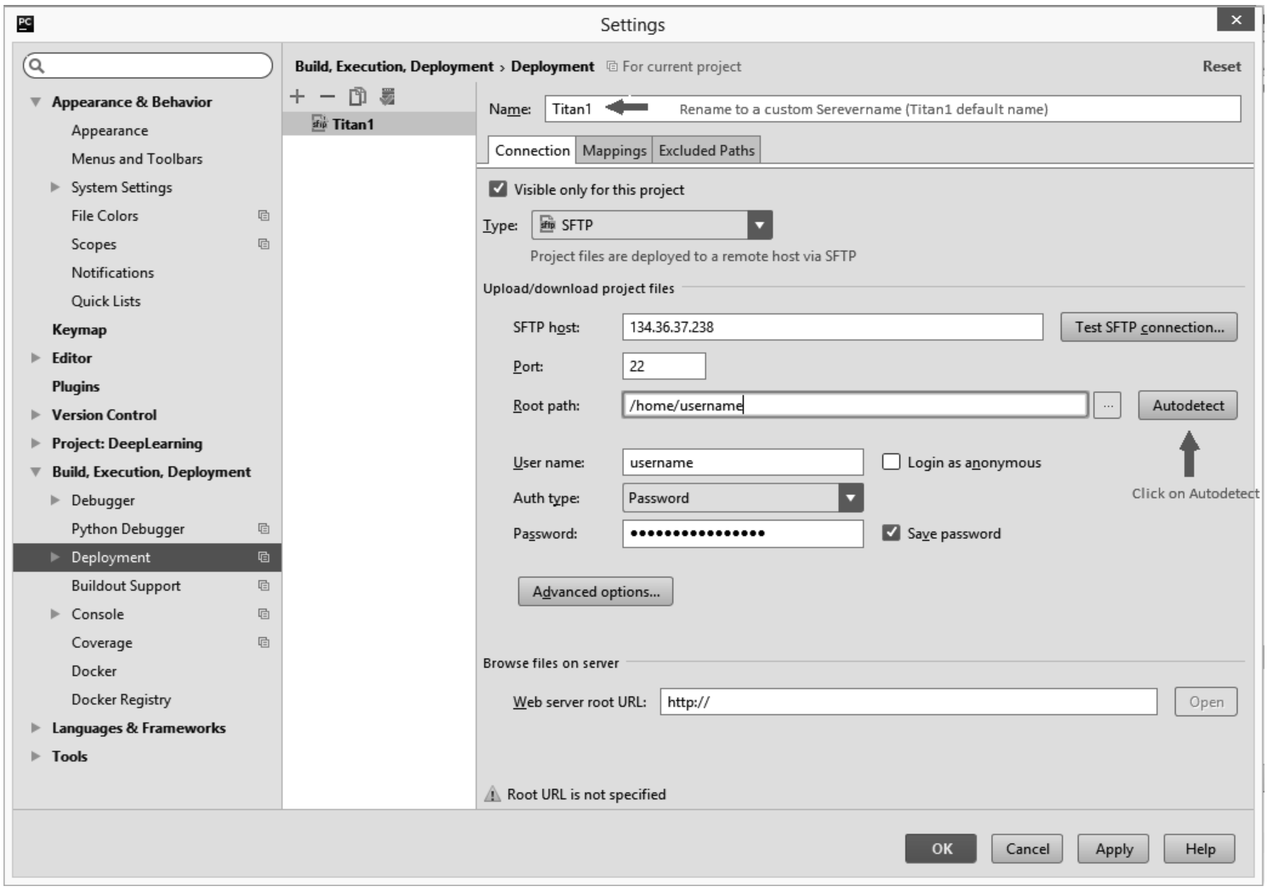
Your machine is now linked to Titan1 server! Now a deployment configuration is needed.

Deployment configuration for data and code

This section is needed to send your local code to the Titan1 server. Is possible to use this system also for data.

Click on create and go to **File/Settings ->** Under Build, Execution, Deployment select Deployment. You should see a server configuration (called *ssh://134.36.37.23…*). Rename the server if you want. Type **must** be SFTP. **Click on Autodetect**.

The result should be similar to this image:



Settings

Build, Execution, Deployment › Deployment For current project

Name: Titan1 Rename to a custom Serevername (Titan1 default name)

Visible only for this project: ✓

Type: SFTP

Project files are deployed to a remote host via SFTP

Upload/download project files

SFTP host: 134.36.37.238

Port: 22

Root path: /home/username

Click on Autodetect

User name: username

Login as anonymous: ☐

Auth type: Password

Password: ••••••••••••••••

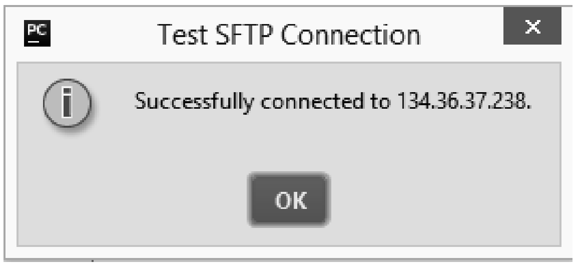
Save password: ✓

Browse files on server

Web server root URL: http://

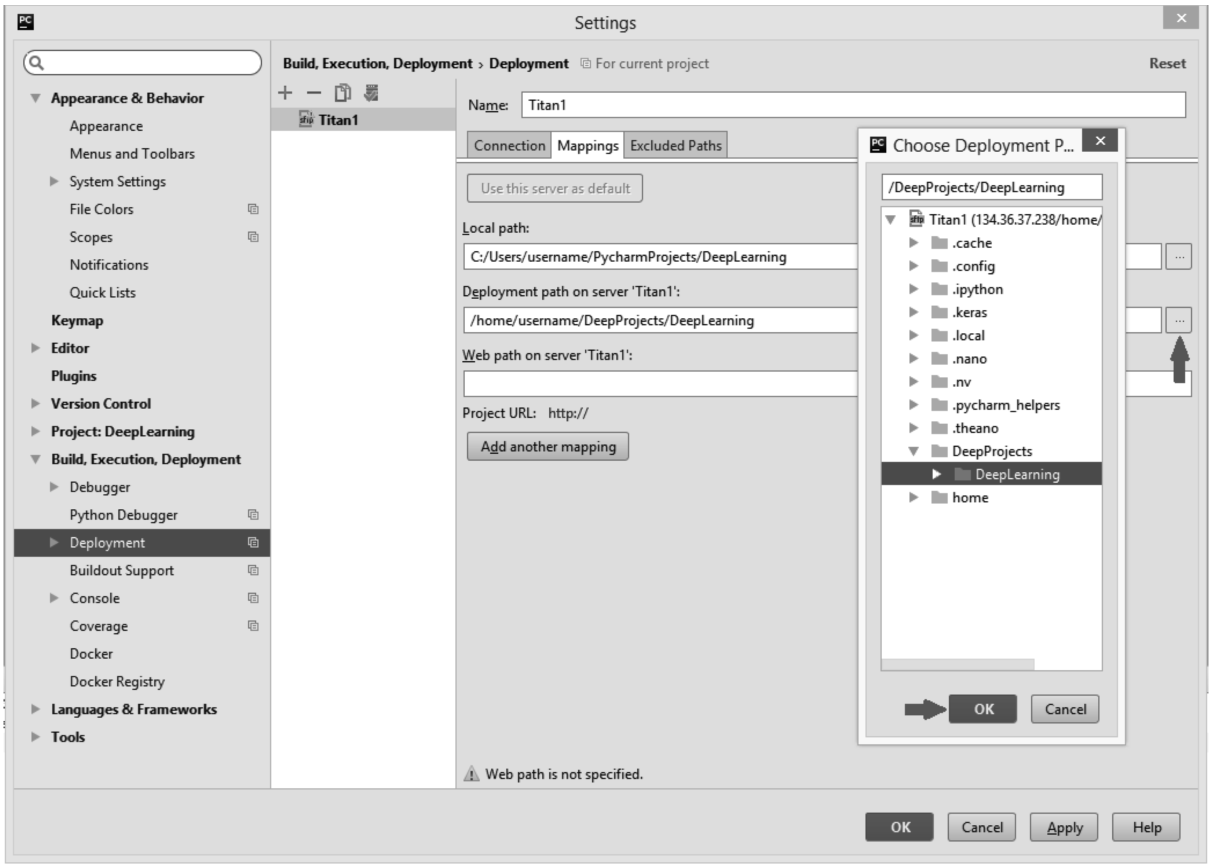
Root URL is not specified

Pressing *Test SFTP connection…* you should see this popup:


Test SFTP Connection

Successfully connected to 134.36.37.238.

Under *Mappings* layer fill Deployment path on server *Titan1* with the path of your project. The path set automatically is not correct because of */home/username/* part. Click on … button and select your folder (<u>with a right click is possible to create a new one</u>).


Settings

Build, Execution, Deployment › Deployment   For current project

Name: Titan1

Local path:
C:/Users/username/PycharmProjects/DeepLearning

Deployment path on server 'Titan1':
/home/username/DeepProjects/DeepLearning

Web path on server 'Titan1':

Project URL: http://

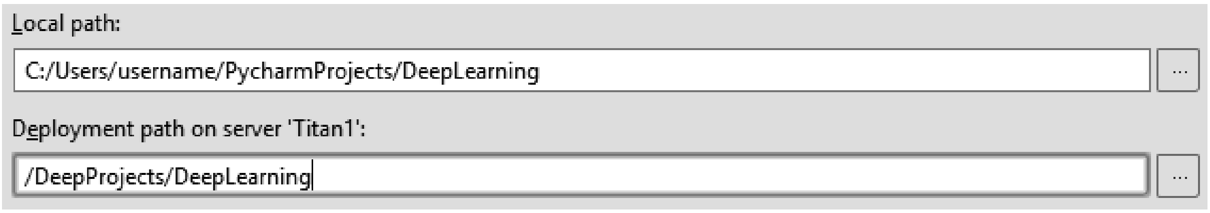
Web path is not specified.

Choose Deployment P…

/DeepProjects/DeepLearning

Titan1 (134.36.37.238/home/

Press OK. The result <u>must be</u> similar to:

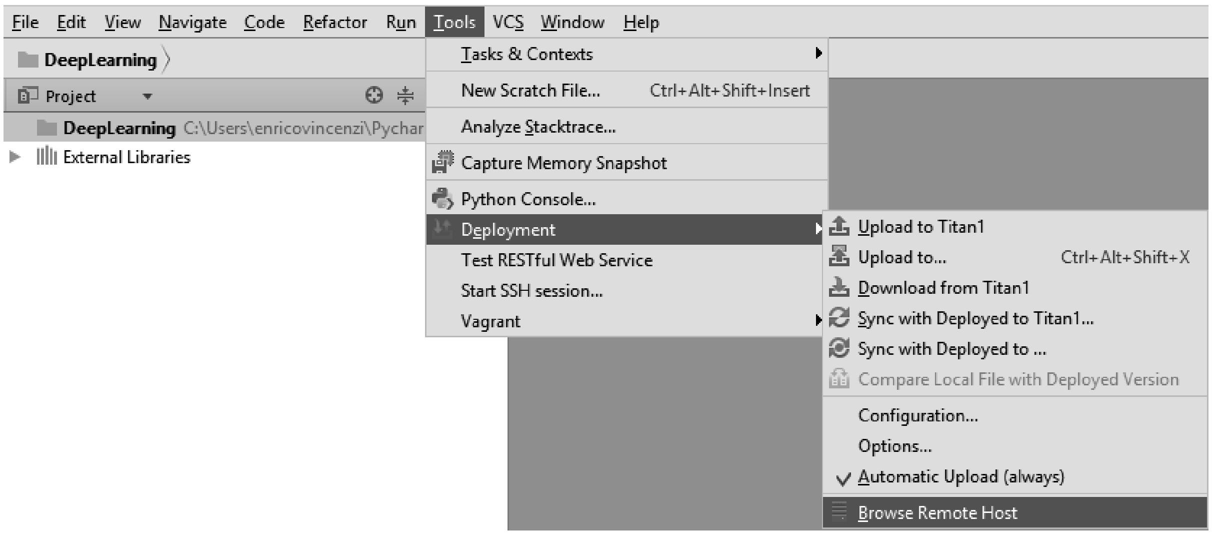

Local path:
C:/Users/username/PycharmProjects/DeepLearning

Deployment path on server 'Titan1':
/DeepProjects/DeepLearning

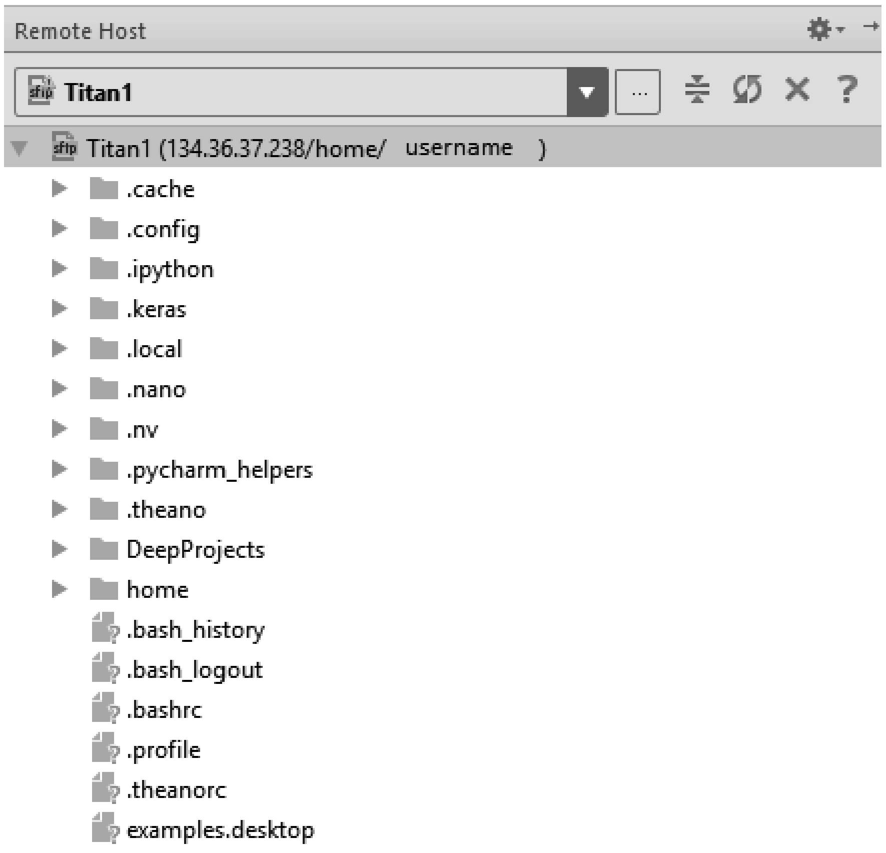
No */home/username/* should be on the path. Press OK.

Remote browser

Click on *Tools -> Deployment -> Browse Remote Host*

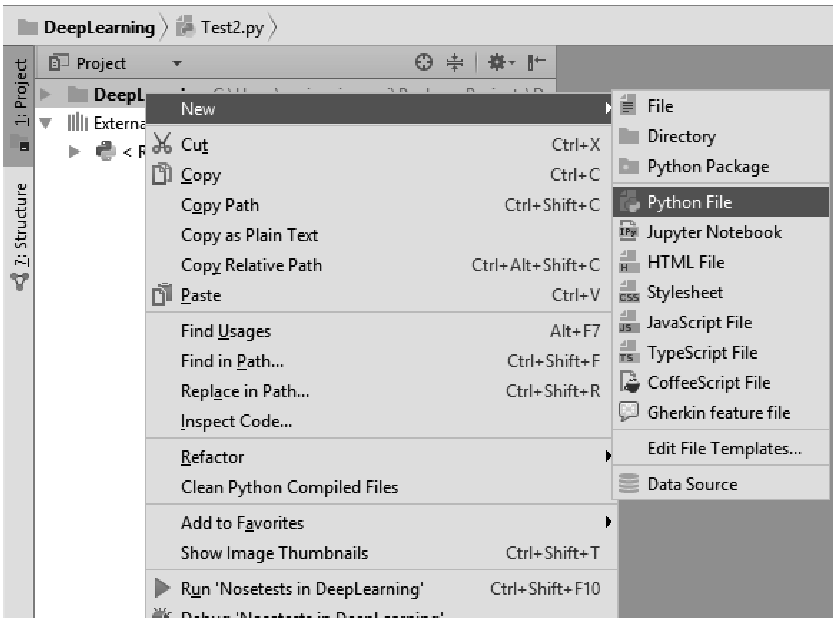


This docked window should appear:



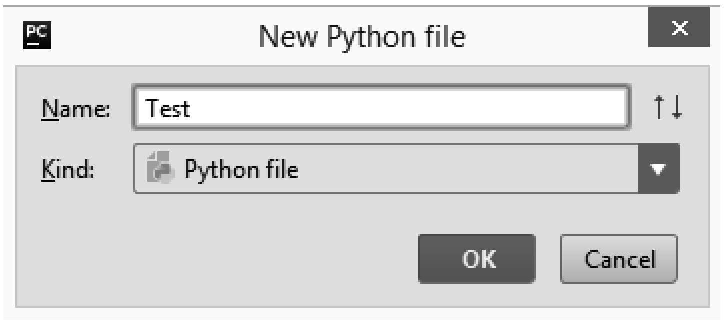
Here is possible to browse and check all files on the remote server.

Create a new Python file

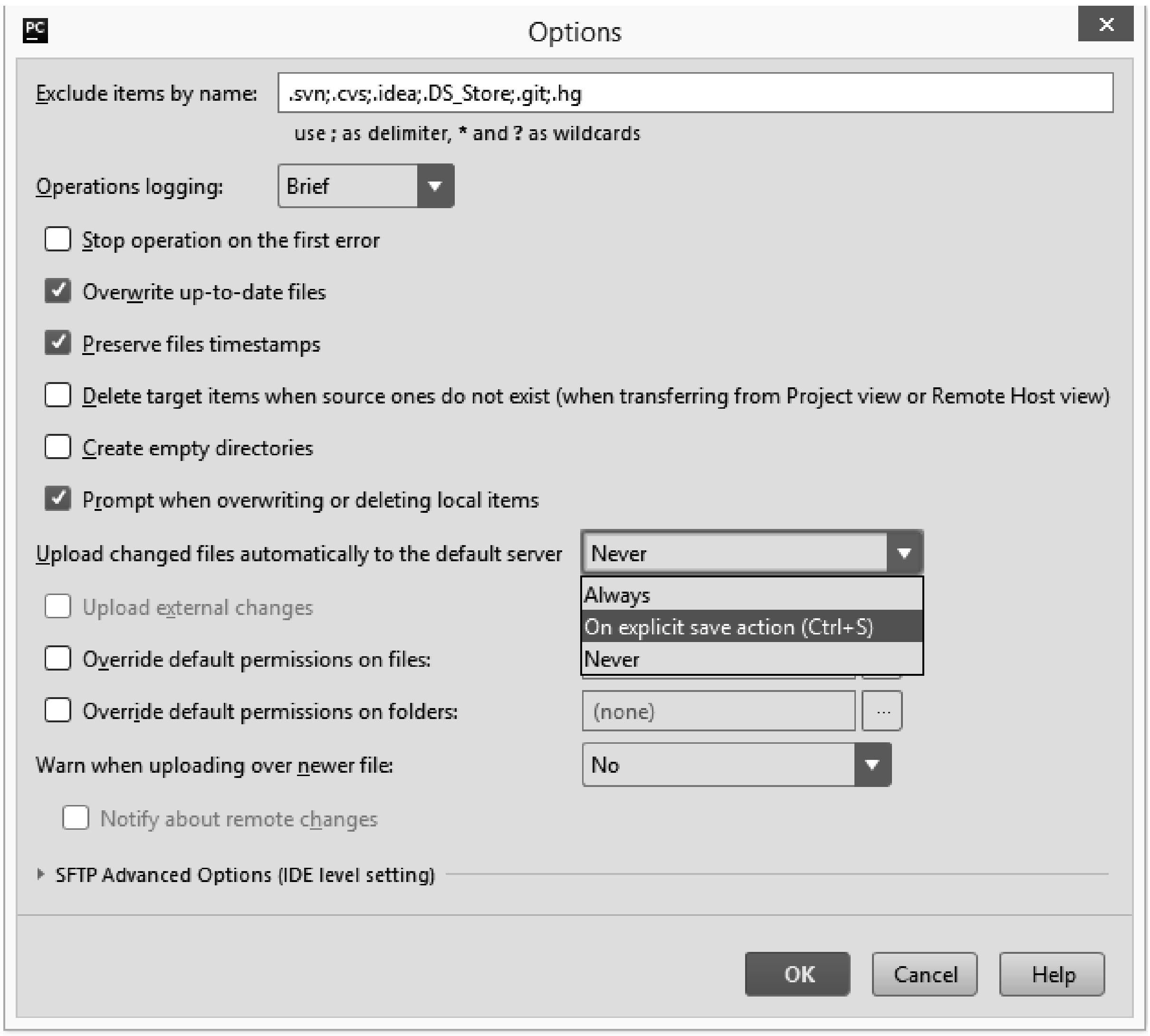


In order to create a new python file:

1. Right click on the project name;
2. New;
3. Python file;
4. Choose an name and press OK.



Save to the server


Options

Exclude items by name: .svn;.cvs;.idea;.DS_Store;.git;.hg

use ; as delimiter, * and ? as wildcards

Operations logging: Brief

☐ Stop operation on the first error

✓ Overwrite up-to-date files

✓ Preserve files timestamps

☐ Delete target items when source ones do not exist (when transferring from Project view or Remote Host view)

☐ Create empty directories

✓ Prompt when overwriting or deleting local items

Upload changed files automatically to the default server: Never

- Always
- On explicit save action (Ctrl+S)
- Never

☐ Upload external changes

☐ Override default permissions on files:

☐ Override default permissions on folders: (none)

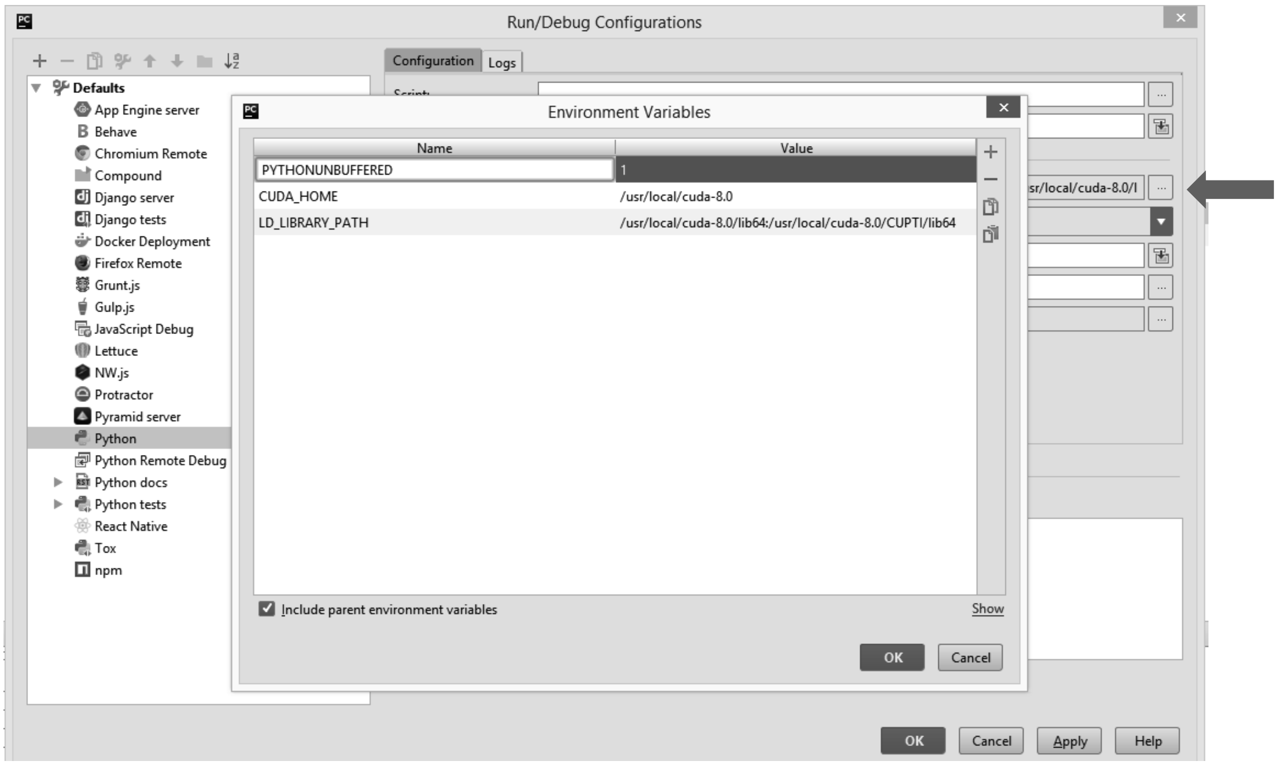
Warn when uploading over newer file: No

☐ Notify about remote changes

SFTP Advanced Options (IDE level setting)

1. From the main bar click on Tools/Deployment -> Options… ;
2. Under '*Upload changed files automatically to the default server'* you can select two options:
    a. *On explicit save action* if you want to send the server only on explicit CTRL + S command;
    b. *Always* if you want a real time upload on each change on local;
3. Press OK.

Run configuration (CUDA path)

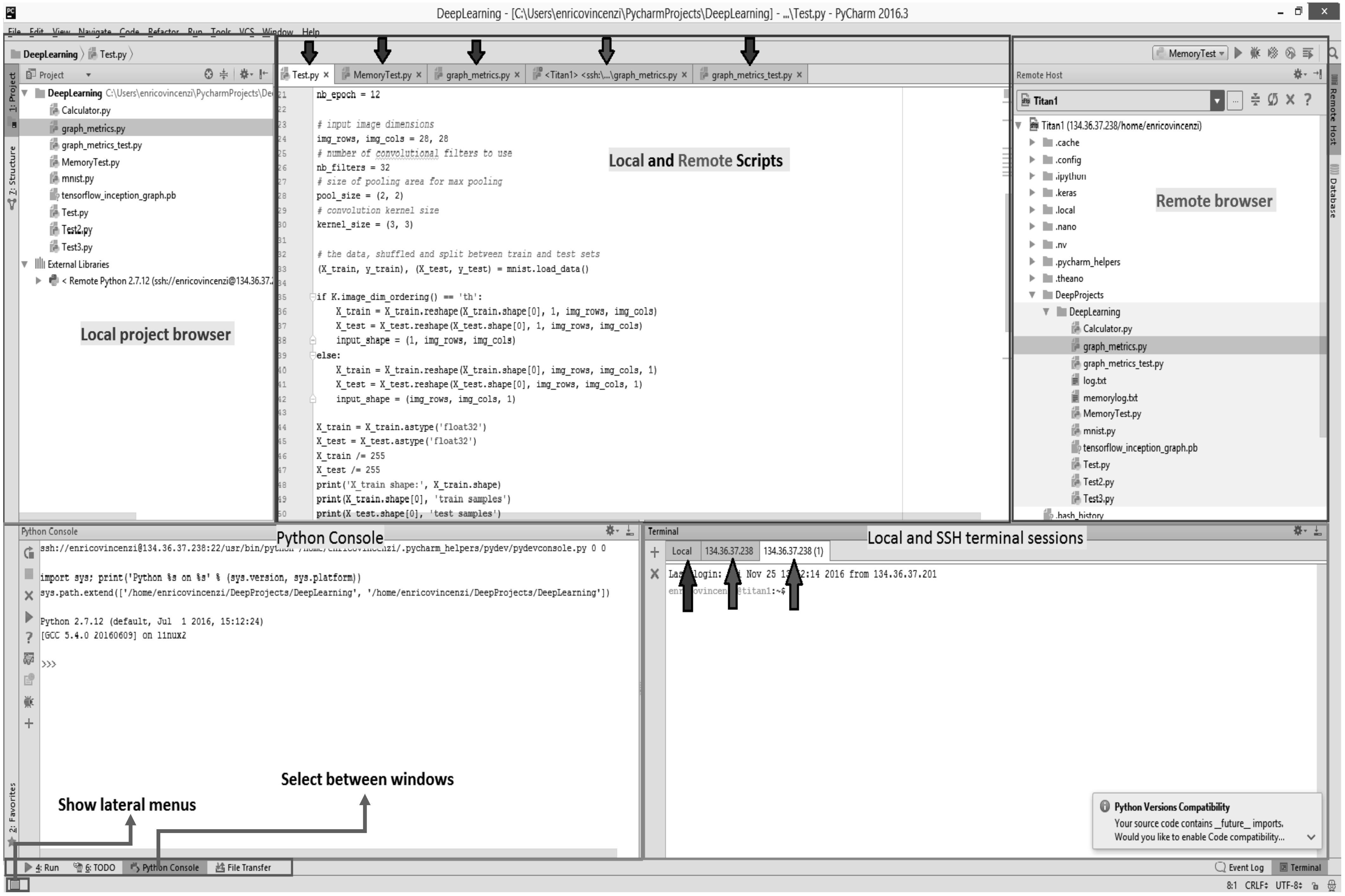


Name	Value
PYTHONUNBUFFERED	1
CUDA_HOME	/usr/local/cuda-8.0
LD_LIBRARY_PATH	/usr/local/cuda-8.0/lib64:/usr/local/cuda-8.0/CUPTI/lib64

In this section will be explained how to avoid CUDA libraries errors with remote interpreter.

1. From the main bar click on *Run -> Edit Configurations…* ;
2. On the left frame **under Defaults** click on Python;
3. In the Configuration tab edit *Environment variables*;
4. Add two new variables through + button:
   a. CUDA_HOME = /usr/local/cuda-8.0
   b. LD_LIBRARY_PATH = /usr/local/cuda-8.0/lib64:/usr/local/cuda-8.0/extras/CUPTI/lib64
5. Click OK;
6. Click OK.

Now is possible to run python script on the server with local output and debug.



```
nb_epoch = 12

# input image dimensions
img_rows, img_cols = 28, 28
# number of convolutional filters to use
nb_filters = 32
# size of pooling area for max pooling
pool_size = (2, 2)
# convolution kernel size
kernel_size = (3, 3)

# the data, shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = mnist.load_data()

if K.image_dim_ordering() == 'th':
    X_train = X_train.reshape(X_train.shape[0], 1, img_rows, img_cols)
    X_test = X_test.reshape(X_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    X_train = X_train.reshape(X_train.shape[0], img_rows, img_cols, 1)
    X_test = X_test.reshape(X_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
print('X_train shape:', X_train.shape)
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')
```

```
ssh://enricovincenzi@134.36.37.238:22/usr/bin/python /home/enricovincenzi/.pycharm_helpers/pydev/pydevconsole.py 0 0

import sys; print('Python %s on %s' % (sys.version, sys.platform))
sys.path.extend(['/home/enricovincenzi/DeepProjects/DeepLearning', '/home/enricovincenzi/DeepProjects/DeepLearning'])

Python 2.7.12 (default, Jul  1 2016, 15:12:24)
[GCC 5.4.0 20160609] on linux2

>>>
```

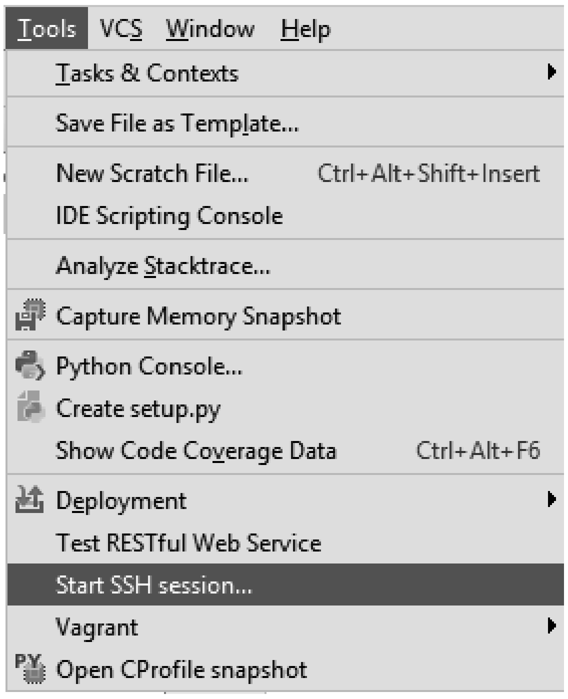
```
Last login: [illegible] Nov 25 1[illegible]:14 2016 from 134.36.37.201
enricovincenzi@titan1:~$
```

Python Versions Compatibility
Your source code contains __future__ imports.
Would you like to enable Code compatibility...

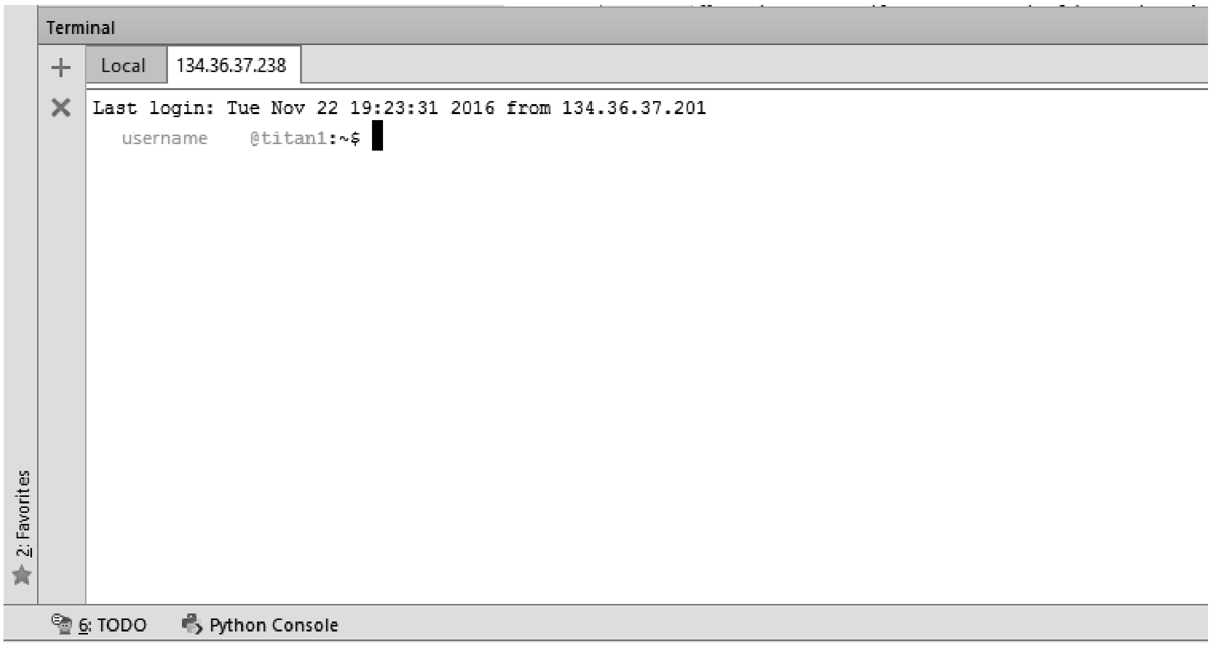
SSH session

For long run session can be useful let the process run in background and disconnect the local machine. For this purpose a normal SSH session is useful.

SSH inside PyCharm requires the first part of this guide.

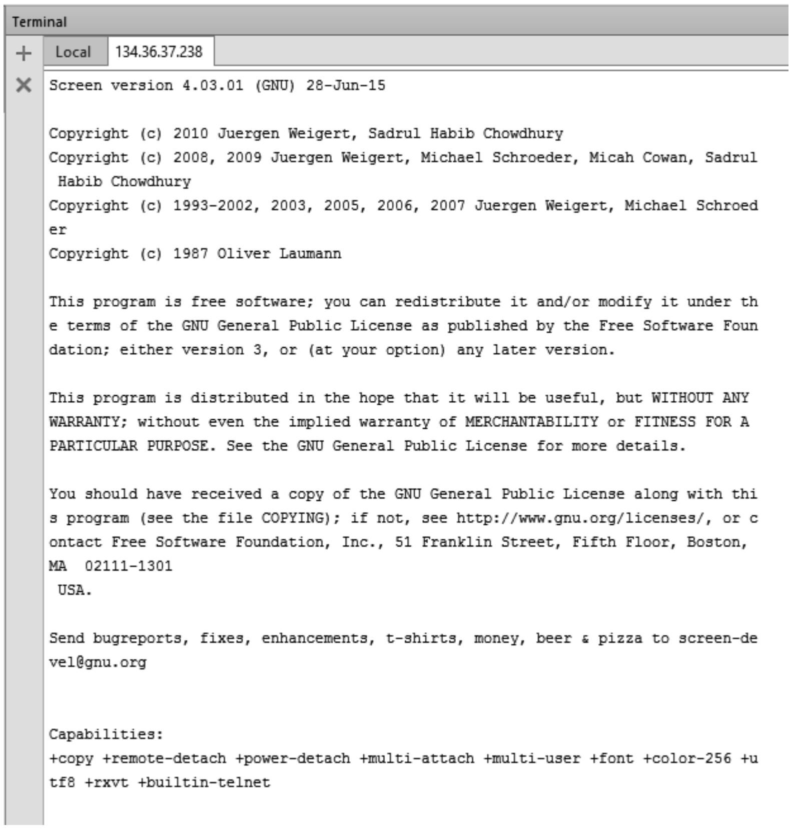


1. From the main bar *Tools/Start SSH session...* ;
2. Select Titan1 server;
3. In the left bottom of the page a SSH session will start;



```
Last login: Tue Nov 22 19:23:31 2016 from 134.36.37.201
  username    @titan1:~$
```

Screen session

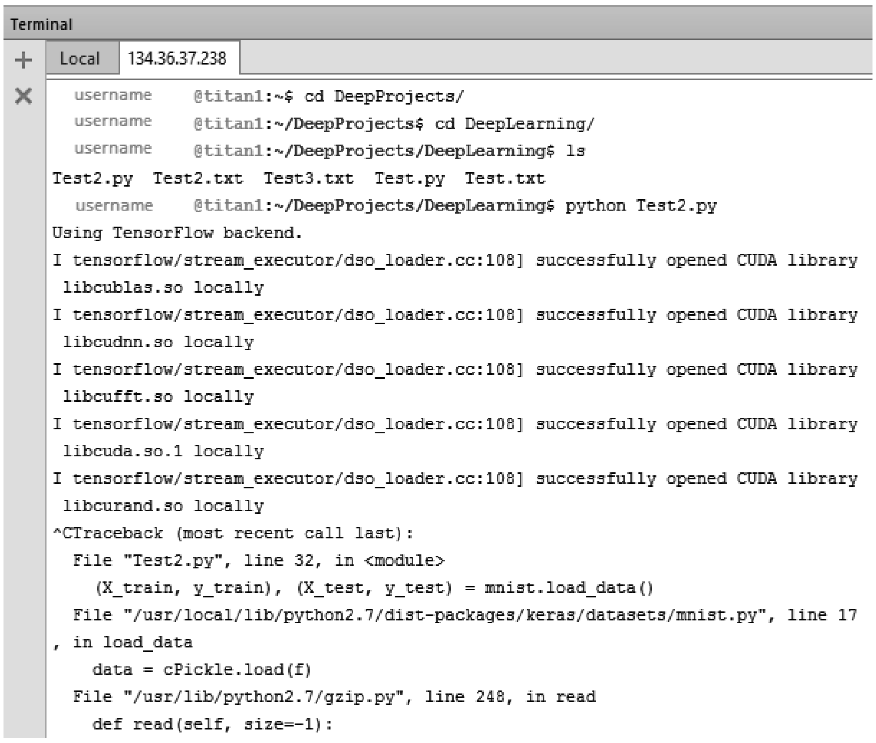


```
Screen version 4.03.01 (GNU) 28-Jun-15

Copyright (c) 2010 Juergen Weigert, Sadrul Habib Chowdhury
Copyright (c) 2008, 2009 Juergen Weigert, Michael Schroeder, Micah Cowan, Sadrul
 Habib Chowdhury
Copyright (c) 1993-2002, 2003, 2005, 2006, 2007 Juergen Weigert, Michael Schroed
er
Copyright (c) 1987 Oliver Laumann

This program is free software; you can redistribute it and/or modify it under th
e terms of the GNU General Public License as published by the Free Software Foun
dation; either version 3, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY
WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with thi
s program (see the file COPYING); if not, see http://www.gnu.org/licenses/, or c
ontact Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
MA  02111-1301
 USA.

Send bugreports, fixes, enhancements, t-shirts, money, beer & pizza to screen-de
vel@gnu.org


Capabilities:
+copy +remote-detach +power-detach +multi-attach +multi-user +font +color-256 +u
tf8 +rxvt +builtin-telnet
```

In order to detach an SSH process (and logout without killing the process) *screen* is used. (Click here for more info about *screen*)

Type screen on SSH window

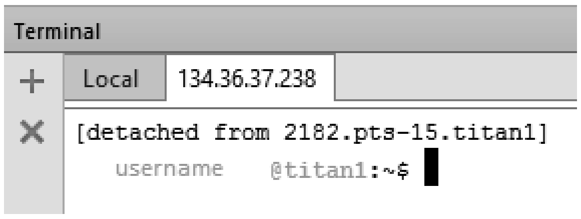
Press Space bar and launch your script.



```
  username    @titan1:~$ cd DeepProjects/
  username    @titan1:~/DeepProjects$ cd DeepLearning/
  username    @titan1:~/DeepProjects/DeepLearning$ ls
Test2.py  Test2.txt  Test3.txt  Test.py  Test.txt
  username    @titan1:~/DeepProjects/DeepLearning$ python Test2.py
Using TensorFlow backend.
I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library
 libcublas.so locally
I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library
 libcudnn.so locally
I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library
 libcufft.so locally
I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library
 libcuda.so.1 locally
I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library
 libcurand.so locally
^CTraceback (most recent call last):
  File "Test2.py", line 32, in <module>
    (X_train, y_train), (X_test, y_test) = mnist.load_data()
  File "/usr/local/lib/python2.7/dist-packages/keras/datasets/mnist.py", line 17
, in load_data
    data = cPickle.load(f)
  File "/usr/lib/python2.7/gzip.py", line 248, in read
    def read(self, size=-1):
```

Detach process

Press CTRL + a and then type d (lowercase) to detach the process.

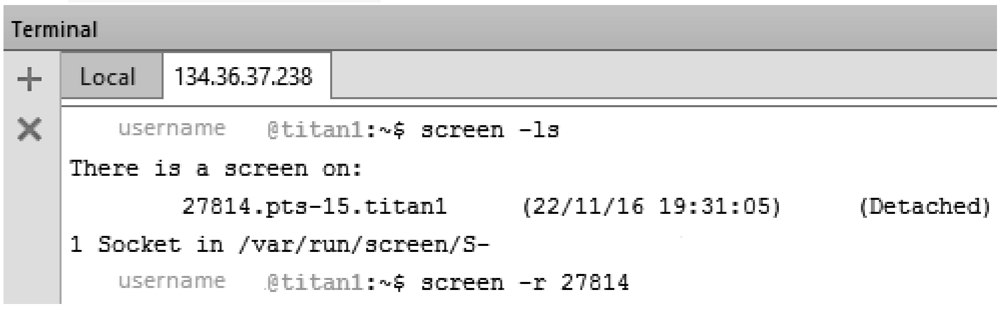


```
[detached from 2182.pts-15.titan1]
  username    @titan1:~$
```

Now is possible to logout and the process will run in background.

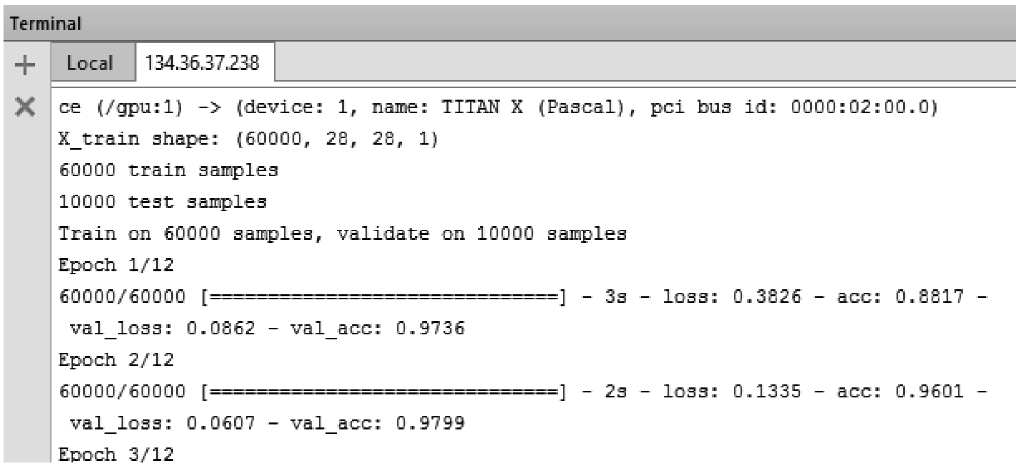
Re-attach process

1. Open a SSH session on Titan1 server;
2. Type *screen -ls*;
3. A list of screen process is enumerated;
4. Type *screen -r 'process number'*.



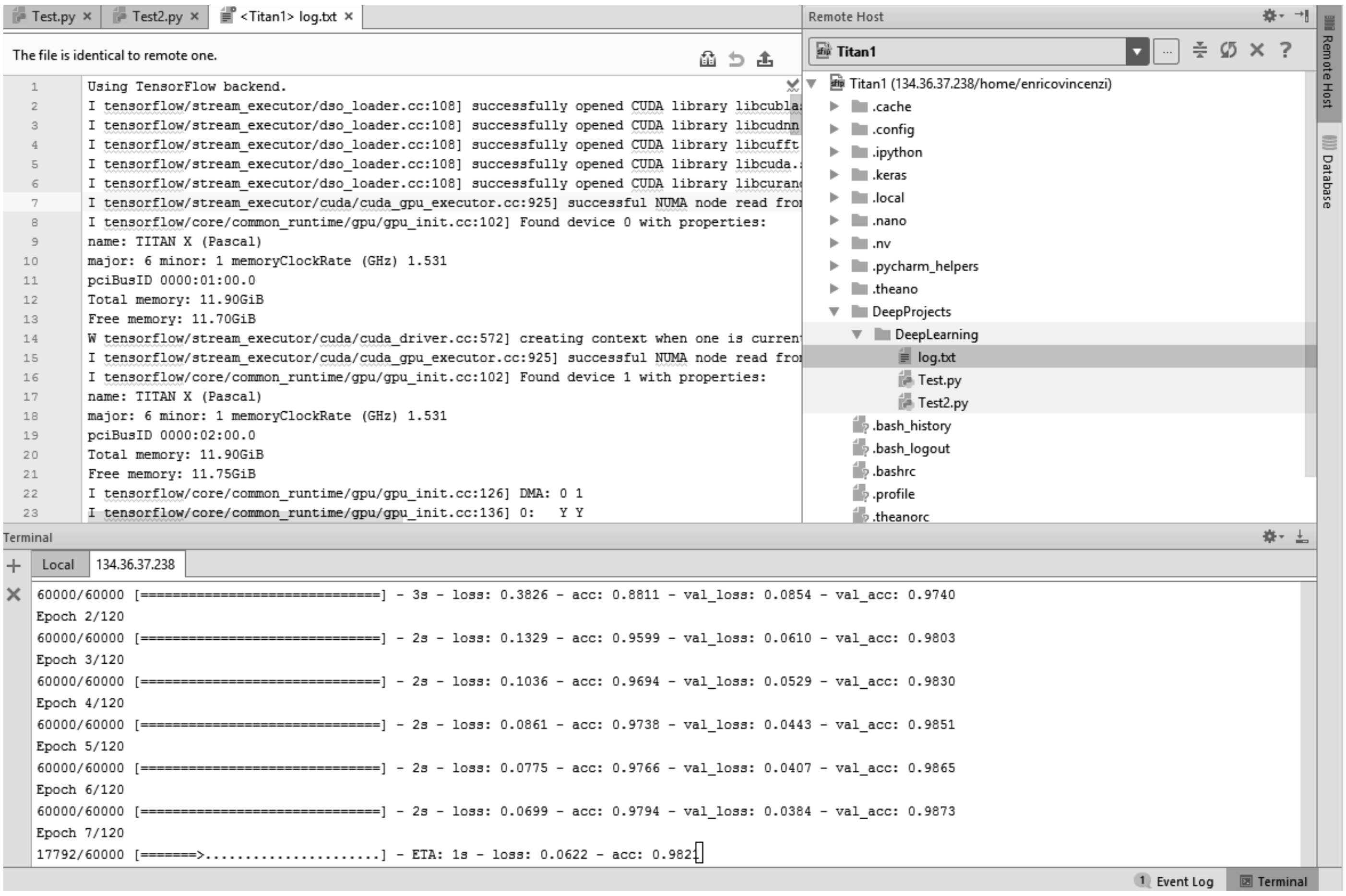
```
    username    @titan1:~$ screen -ls
There is a screen on:
        27814.pts-15.titan1     (22/11/16 19:31:05)     (Detached)
1 Socket in /var/run/screen/S-
    username    @titan1:~$ screen -r 27814
```

5. Process is re-attached.



```
ce (/gpu:1) -> (device: 1, name: TITAN X (Pascal), pci bus id: 0000:02:00.0)
X_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
Train on 60000 samples, validate on 10000 samples
Epoch 1/12
60000/60000 [==============================] - 3s - loss: 0.3826 - acc: 0.8817 -
 val_loss: 0.0862 - val_acc: 0.9736
Epoch 2/12
60000/60000 [==============================] - 2s - loss: 0.1335 - acc: 0.9601 -
 val_loss: 0.0607 - val_acc: 0.9799
Epoch 3/12
```

Tee: save log on file


Test.py × Test2.py × <Titan1> log.txt ×

The file is identical to remote one.

```
Using TensorFlow backend.
I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library libcubla
I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library libcudnn
I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library libcufft
I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library libcuda.
I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library libcuran
I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:925] successful NUMA node read fro
I tensorflow/core/common_runtime/gpu/gpu_init.cc:102] Found device 0 with properties:
name: TITAN X (Pascal)
major: 6 minor: 1 memoryClockRate (GHz) 1.531
pciBusID 0000:01:00.0
Total memory: 11.90GiB
Free memory: 11.70GiB
W tensorflow/stream_executor/cuda/cuda_driver.cc:572] creating context when one is curren
I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:925] successful NUMA node read fro
I tensorflow/core/common_runtime/gpu/gpu_init.cc:102] Found device 1 with properties:
name: TITAN X (Pascal)
major: 6 minor: 1 memoryClockRate (GHz) 1.531
pciBusID 0000:02:00.0
Total memory: 11.90GiB
Free memory: 11.75GiB
I tensorflow/core/common_runtime/gpu/gpu_init.cc:126] DMA: 0 1
I tensorflow/core/common_runtime/gpu/gpu_init.cc:136] 0:   Y Y
```

Remote Host

Titan1

```
Titan1 (134.36.37.238/home/enricovincenzi)
  .cache
  .config
  .ipython
  .keras
  .local
  .nano
  .nv
  .pycharm_helpers
  .theano
  DeepProjects
    DeepLearning
      log.txt
      Test.py
      Test2.py
  .bash_history
  .bash_logout
  .bashrc
  .profile
  .theanorc
```

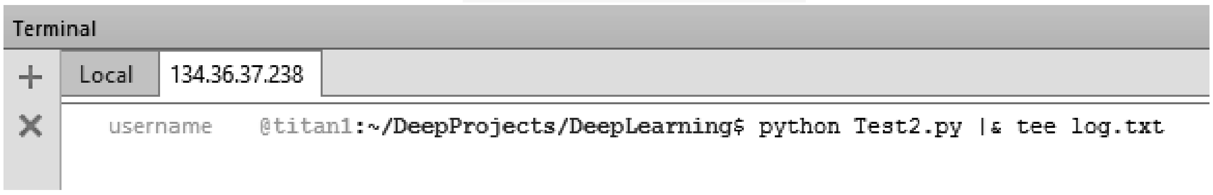
Terminal

Local | 134.36.37.238

```
60000/60000 [==============================] - 3s - loss: 0.3826 - acc: 0.8811 - val_loss: 0.0854 - val_acc: 0.9740
Epoch 2/120
60000/60000 [==============================] - 2s - loss: 0.1329 - acc: 0.9599 - val_loss: 0.0610 - val_acc: 0.9803
Epoch 3/120
60000/60000 [==============================] - 2s - loss: 0.1036 - acc: 0.9694 - val_loss: 0.0529 - val_acc: 0.9830
Epoch 4/120
60000/60000 [==============================] - 2s - loss: 0.0861 - acc: 0.9738 - val_loss: 0.0443 - val_acc: 0.9851
Epoch 5/120
60000/60000 [==============================] - 2s - loss: 0.0775 - acc: 0.9766 - val_loss: 0.0407 - val_acc: 0.9865
Epoch 6/120
60000/60000 [==============================] - 2s - loss: 0.0699 - acc: 0.9794 - val_loss: 0.0384 - val_acc: 0.9873
Epoch 7/120
17792/60000 [=======>......................] - ETA: 1s - loss: 0.0622 - acc: 0.9821
```

Can be useful save run log on file. Tee is a command that allows to save all console log on a file without redirect STDOUT and STDERR.

1. Start a SSH session to Titan1 server;
2. Launch Screen;
3. Launch your python script in this way: *python Test.py |& tee log.txt* ;


Terminal

Local | 134.36.37.238

```
username    @titan1:~/DeepProjects/DeepLearning$ python Test2.py |& tee log.txt
```

4. Log of STDOUT and STDERR are saved on log.txt;

Omit & symbol to drop STDERR log from the file.

<u>Both STDOUT and STDERR are still visualized as output on the console</u> as shown in the first image of this page.

Example code for testing

In order to test the configuration run classic *Keras mnist* example (copy the code from the link):

```
'''Trains a simple convnet on the MNIST dataset.
Gets to 99.25% test accuracy after 12 epochs
(there is still a lot of margin for parameter tuning).
16 seconds per epoch on a GRID K520 GPU.
'''

from __future__ import print_function
import numpy as np
np.random.seed(1337)  # for reproducibility

from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D
from keras.utils import np_utils
from keras import backend as K

batch_size = 128
nb_classes = 10
nb_epoch = 12

# input image dimensions
img_rows, img_cols = 28, 28
# number of convolutional filters to use
nb_filters = 32
# size of pooling area for max pooling
pool_size = (2, 2)
# convolution kernel size
kernel_size = (3, 3)

# the data, shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = mnist.load_data()

if K.image_dim_ordering() == 'th':
    X_train = X_train.reshape(X_train.shape[0], 1, img_rows, img_cols)
    X_test = X_test.reshape(X_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    X_train = X_train.reshape(X_train.shape[0], img_rows, img_cols, 1)
    X_test = X_test.reshape(X_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
print('X_train shape:', X_train.shape)
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')

# convert class vectors to binary class matrices
Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)

model = Sequential()

model.add(Convolution2D(nb_filters, kernel_size[0], kernel_size[1],
                        border_mode='valid',
                        input_shape=input_shape))
model.add(Activation('relu'))
model.add(Convolution2D(nb_filters, kernel_size[0], kernel_size[1]))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=pool_size))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(128))
model.add(Activation('relu'))
```

4.2.2 Lua

Lua environment, used by Torch, is less known than Python one. The reason why has been used in this work is that the original work for Grad-CAM visualization technique is implemented in Torch. Moreover, Torch is very useful and, within the author personal opinion, easier to read and learn than Tensorflow. The main drawback is just the lack of the same number of image analysis libraries and IDEs. The IDE used is ZeroBraneStudio, the only one, at March 2017, with a sufficiently stable debugger.

4.3 Datasets

As anticipated the dataset is the one provided by EyePACS for Diabetic Retinopathy Detection Kaggle challenge. The dataset is a 5-class dataset with 90000 high resolution, labeled, fundus retinal scans. The resolution used in the experiments is 300×300 and all the images satisfy this constraint. The images are divided in the different classes with the respect of the diabetic retinopathy diagnosed:

1. NO Diabetic Retinopathy;
2. Mild Diabetic Retinopathy;
3. Moderate Diabetic Retinopathy;
4. Severe Diabetic Retinopathy;
5. Proliferative Diabetic Retinopathy.

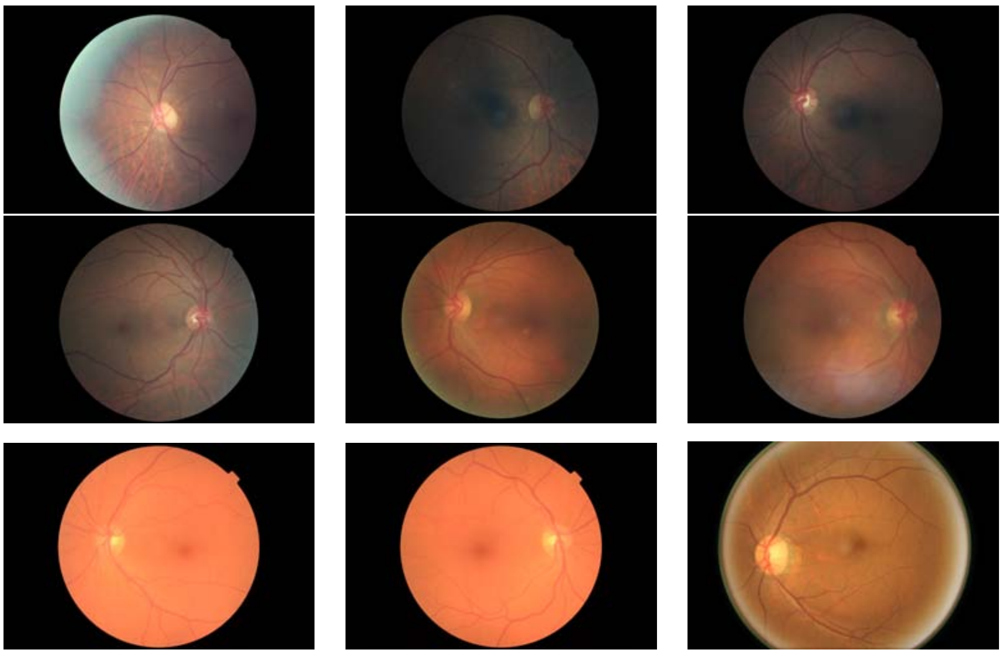


Figure 4.1: NO Diabetic Retinopathy

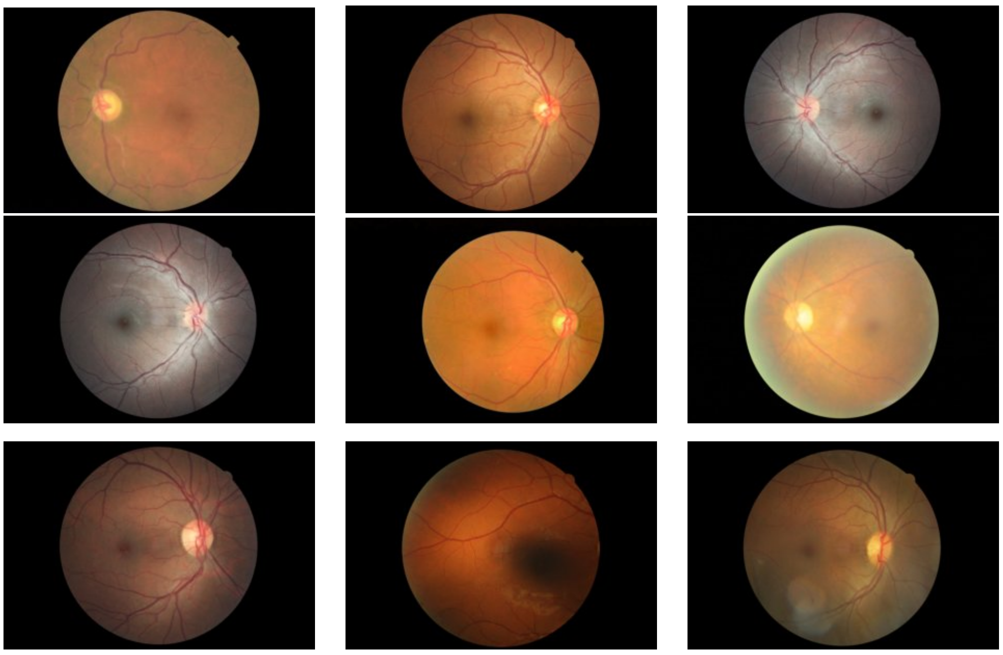


Figure 4.2: Mild Diabetic Retinopathy

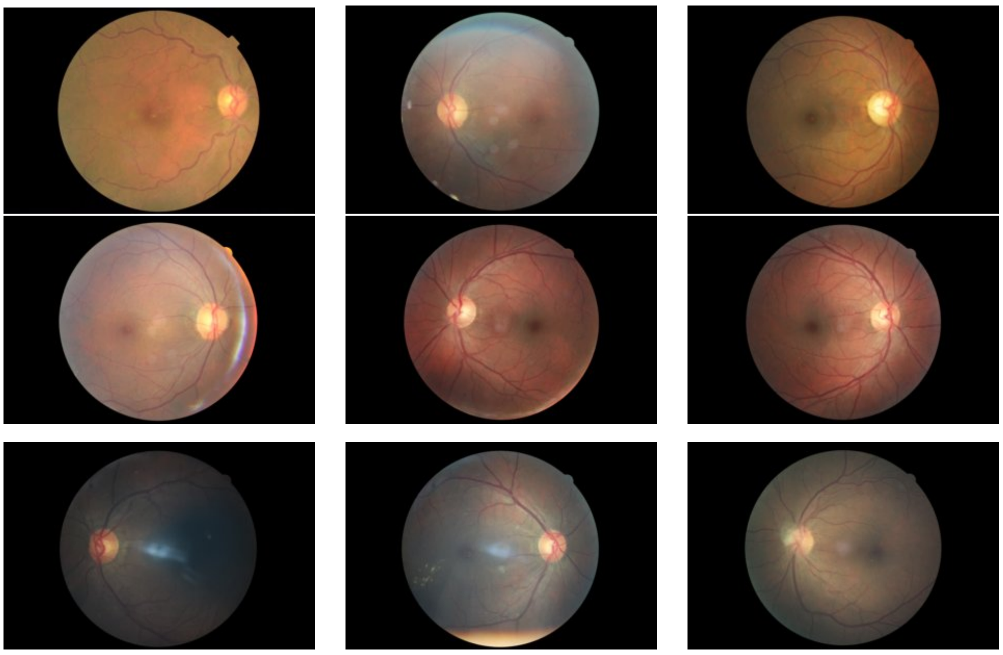


Figure 4.3: Moderate Diabetic Retinopathy

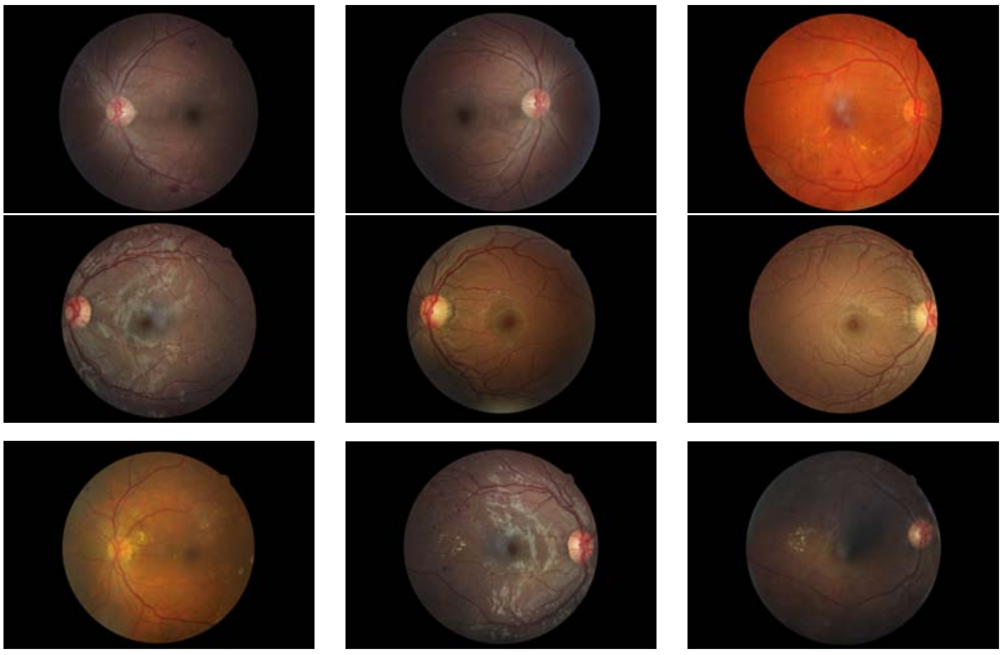


Figure 4.4: Severe Diabetic Retinopathy

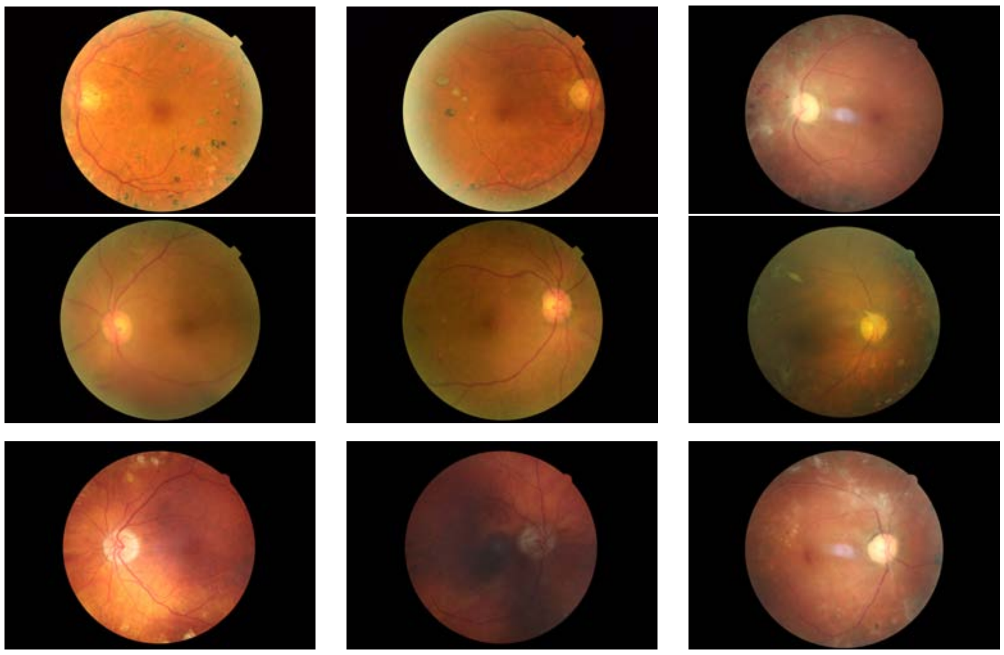


Figure 4.5: Proliferative Diabetic Retinopathy

The images are taken with different cameras and at different resolution and light condition. A normalization between the images can be a good idea to reduce the noise given by this factors.

4.3.1 Preprocessing

The preprocessing used to normalise images is the following:

$$O = (I - Gaussian(I)) + 128 \tag{4.1}$$

where O is the output image, I the input image and the 128 bias is used to normalise the image around the middle value in a *uint* image representation (code in Appendix 2).

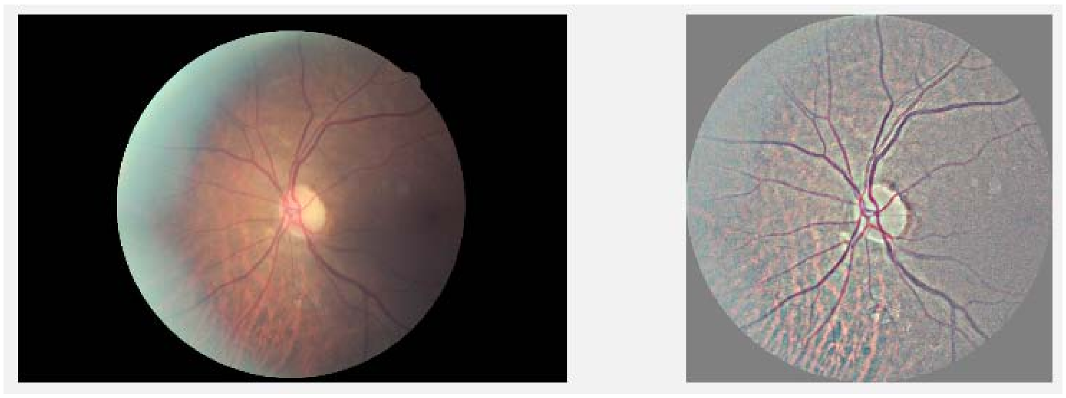


Figure 4.6: Preprocessing Input and Output example

4.3.2 Artificial markers

In order to validate the visualizations techniques will be necessary to have datasets with visible markers. This markers can be natural images or different textures. Each of these *artificial* datasets are composed of 35000 fundus images. On each of these images an artificial marker is added. The code used print an artificial marker with random dimension rotation and position inside the images (code in Appendix 3).

Multiclass: texture markers

The first artificial dataset is a 4-class dataset with:

1. natural fundus images;
2. grass texture blob randomly added to the image;
3. granit texture blob randomly added to the image;
4. fabric texture blob randomly added to the image;

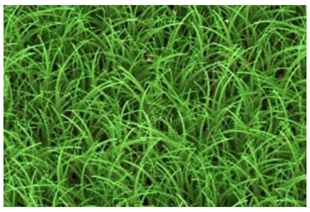


Figure 4.7: Grass

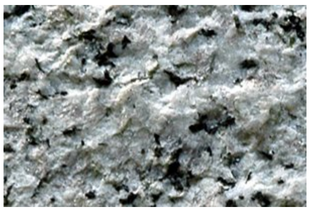


Figure 4.8: Granit

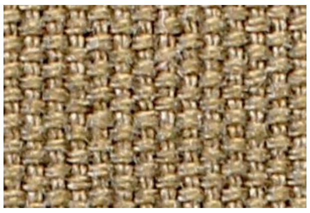


Figure 4.9: Frabric

The output are images with three different texture blobs.

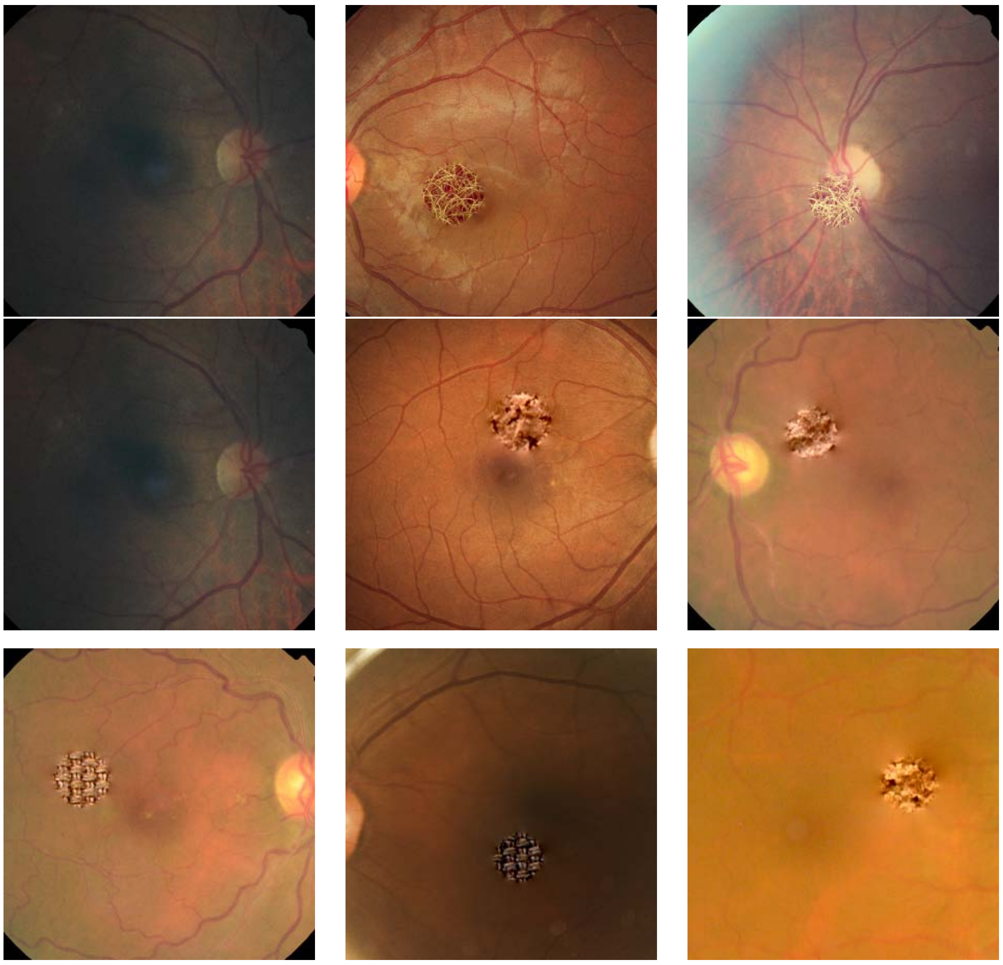


Figure 4.10: Artificial texture markers images samples. The images can have no blob or granit, grass, fabric texture.

Multiclass: natural artificial markers

The first artificial dataset is a 4-class dataset with:

1. grass texture blob randomly added to the image;
2. car model randomly added to the image;
3. droid model randomly added to the image;
4. giraffe randomly added to the image;

5. cartoon boy randomly added to the image;

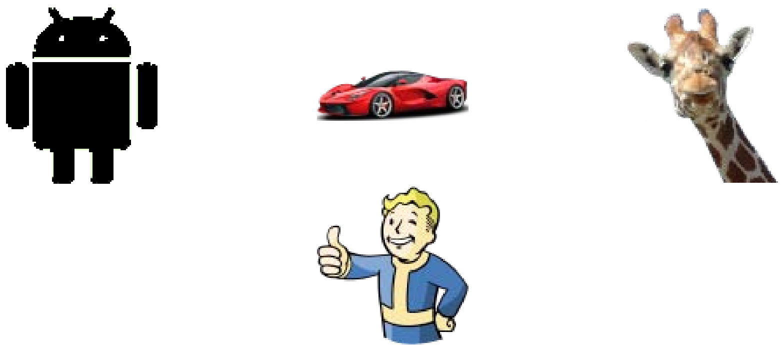


Figure 4.11: Natural models used as markers

Dataset samples:

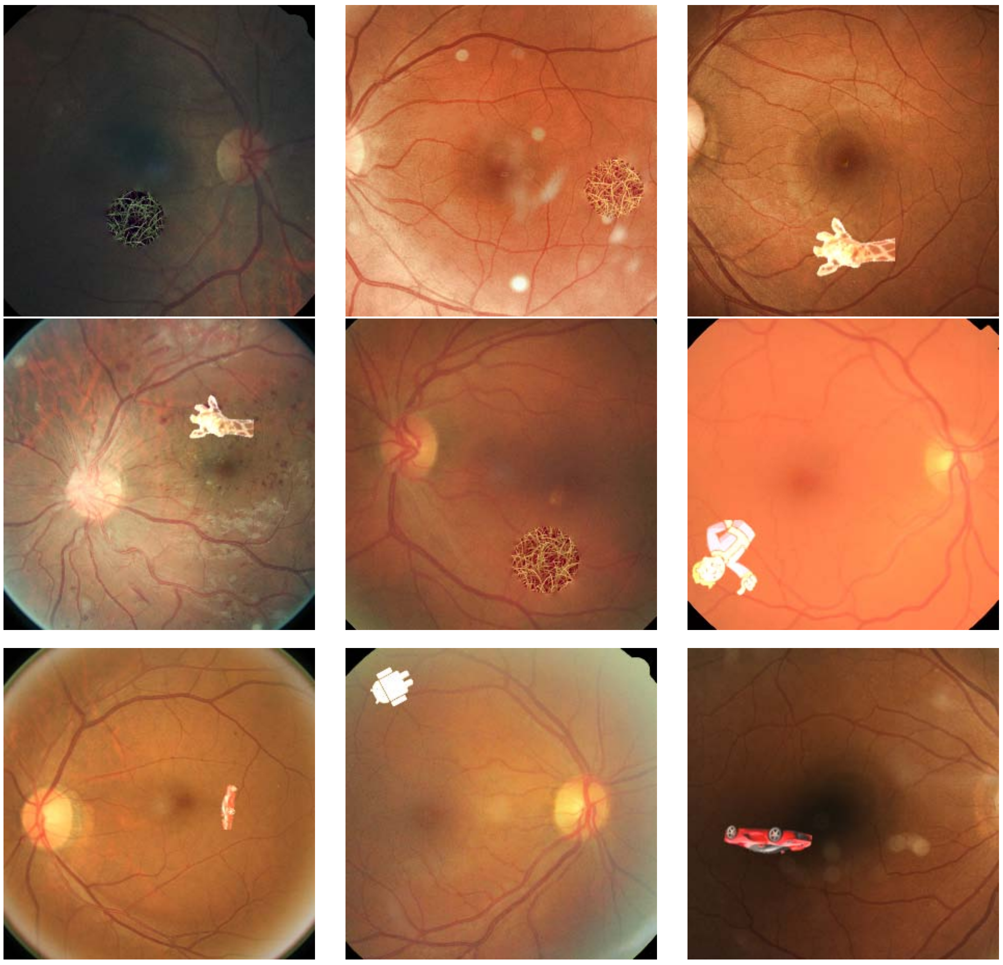


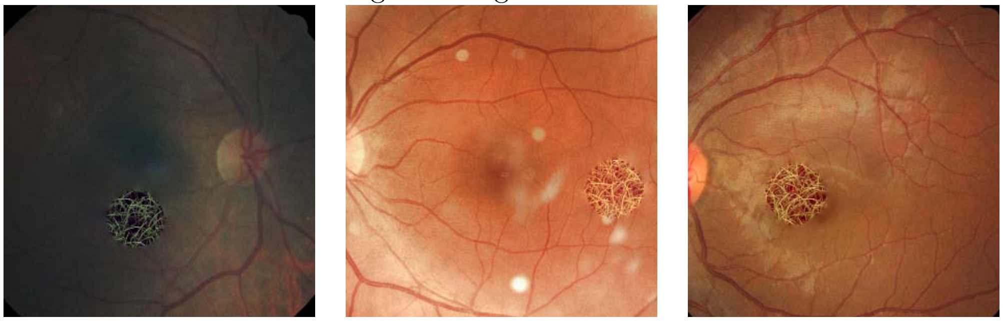
Figure 4.12: Artificial texture markers images samples.

Binary: car , grass blob

The third dataset with artificial is composed by two classes. Both are fundus images, the first with a car model randomly added, and the second with a grass blob.

Binary: grass blob, original fundus images

The last dataset is a binary dataset with non modified fundus images and fundus images with grass texture blob in.

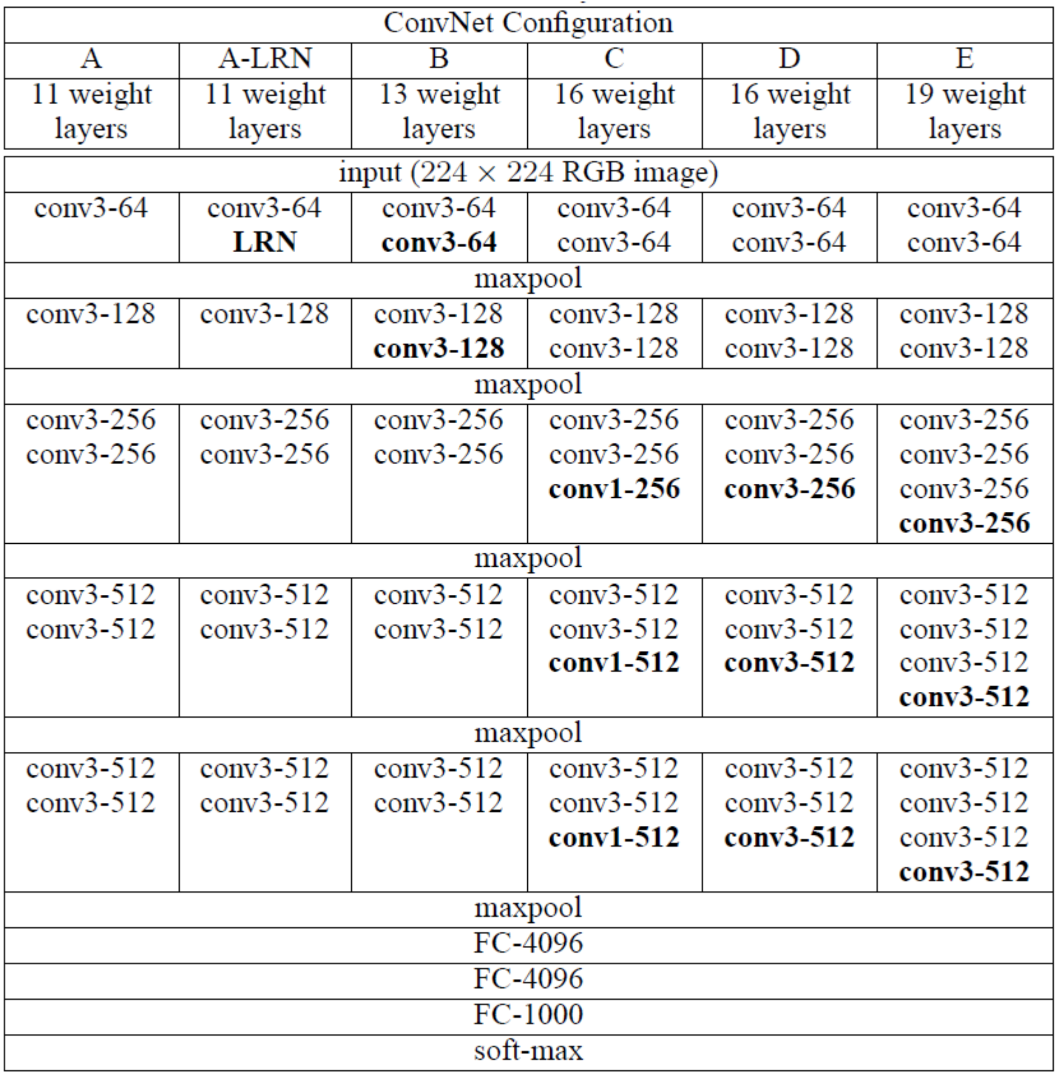


4.4 Network architecure

Classification and, therefore, visualization performances are highly dependent on how the network is designed. Almost infinite combination of layers and parameters can be chosen and the performances are highly dependent on this choice. In this work two architecture designed for natural images classification are taken in account.

VGG

The VGG ([28]), from the name of the development team, has been designed for ILSVRC 2013 challenge and is optimized for multiclass natural images classification.



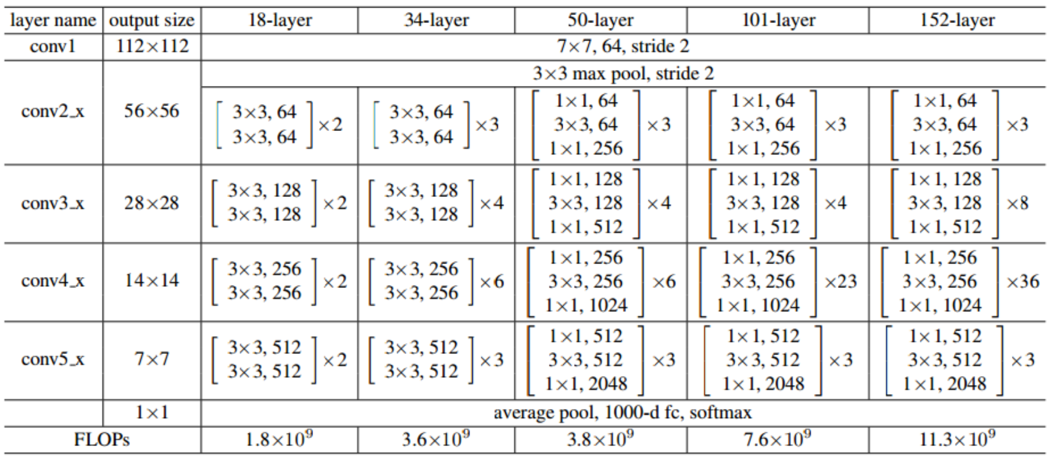
ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 4.13: VGG architecture. D version is the one implemented.

In the table the first row shows all the convolutional layers from left to right; the last part is a fully connected network where N is the umber of classes in the dataset.

ResNet

The ResNet ([29]) has been introduced for ILSVRC 2015 challenge. Is the state of the art for natural image classification. The main difference if compared with the VGG is the non linearity in the architecture design; The reason behind its main performances is a feed-forward stage that subtract the output of each block with its input. This reduce vanishing gradient problem and allows up to 1000 layers architecture. In this work has been used a 50 layers ResNet.



layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times23$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 4.14: ResNet architecture

Where each block is designed as following:

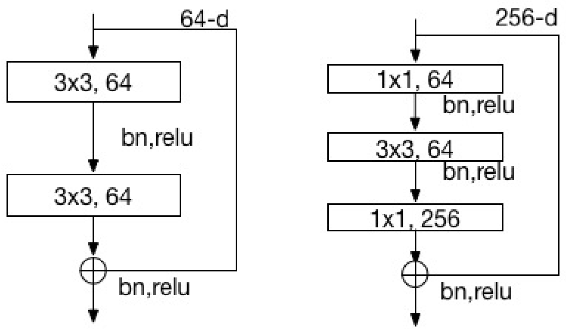


Figure 4.15: ResNet blocks

The main problem of the ResNet is its non sequential design. This make difficult the use of a visualization technique given that the latter requires particular backpropagation steps across the network.

4.5 Visualization techniques

The main aim of this work is visualise what the network can learn and if this knowledge can be used for biomarkers searching and discovery. Two different techniques have been analysed:

- Filter visualisation;
- Gradient Class Activation Mapping (Grad-CAM) ([30], [31]).

4.5.1 Filters visualisation

The first technique is the simplest and more immediate one. The idea is to visualise the filters learned during the training step for each layer. The shallower layers have simplest filters, while deep ones should have more complex ones. Those filters can be useful for visualise if exist some specific pattern useful for biomarkers discovery inside the original image, especially in the deeper layers. Moreover the filters are always useful to understand better training stage and possible Grad-CAM troubles.

4.5.2 Grad-CAM

Grad-CAM ([30], [31]) algorithm is designed to visualise the most important region used by the network for the classification. The input of the algorithm is the trained network, the input image, a number i, with $0 < i <$ # of classes, and the layer of the network from where compute the activation. The output is a heat-map image where each pixel is coloured in a range between red and blue. The higher activation areas are the red one, while the less important are blue. This technique is not dependent on the network architecture type but work better with sequential models like VGG. For this reason Grad-CAM has been implemented and tested mainly with VGG model. ResNet achieves better result than VGG in Imagenet classification challenge, but the main aim of this work is not have a good classification result. To answer the initial question is important to test this technique with medical datasets. ResNet can introduce further issues that are not in the focus of this work.

The original code for this algorithm is deployed in Torch. Let see the details of the code.

Algorithm 2 Grad-CAM verbose algorithm

```
1: Load neural network model and chosen layer name.
2: Remove last activation layer (last tensor in the stack, usually a softmax).
   ▷ An image compatible with the network and representing objects that the network can classify is loaded
3: Load image
4: Set the object class to visualise, or predict the class from the loaded image.
5: doutput = n elements vector (where n is the number of classes) with all values set to 0 but the one related to the chosen class.
   ▷ Grad-CAM steps
   ▷ Two new sequential models are created: model1 and model2
6: model1 = original model from input to chosen layer included.
7: model2 = original model from chosen layer non included to last layer.
   ▷ Weights computation
8: A backward step is computed from doutput to model1.output through model2.
9: activation = the tensor with the weights of the chosen layer.
10: gradients = $\frac{dIn}{dOut}$ where $In$ is the input of model2 and $Out$ its output (all non chosen class positions are set to 0)
11: weight = sum of all the gradients for each activation in the chosen layer.
   ▷ Map computation
12: map = elementwise multiplication between activation and relative weight(previously computed).
13: map = map with all negative values set to 0.
```

```
local model1, model2 = nn.Sequential(), nn.Sequential()
for i = 1, #cnn.modules do
layer_id = tonumber(layer_name)
for i = 1, #cnn.modules do
   model1:add(cnn:get(i))
       if i == layer_id then
          break
       end
end
for i = layer_id+1, #cnn.modules do
   model2:add(cnn:get(i))
end
```

Listing 4.1: model1, model2 setup

The first section of the code initialises model1 and model2 as described in the algorithm.

```
-- Get activations and gradients
model2:zeroGradParameters()
model2:backward(model1.output, doutput)
```

Listing 4.2: backward step in model2

The second section set to 0 all the gradients in model2 and perform back-propagation algorithm from doutput vector layer and the output of model1, used as input for model2.

```
-- Get the activations from model1 and and gradients from model2
local activations = model1.output:squeeze()
local gradients = model2.gradInput:squeeze()
```

Listing 4.3: activations and gradients computation.

The third section define activations variable, the tensor with the wieghts of the previously selected layer to visualise. Then gradients is defined as the derivative of the output of model2 with its input (the layer selected for visualization). The output of model2 is doutput vector. All the values of doutput are set to 0, but the class selected to be visualised.

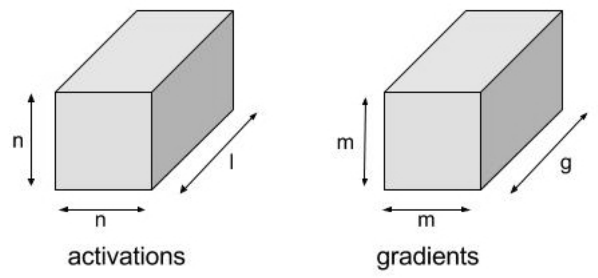


Figure 4.16: Activations and gradients tensor shape. Activation shape = $\{\ell \times n \times n\}$, Gradient shape = $\{g \times m \times m\}$

```
-- Global average pool gradients
local weights = torch.sum(gradients:view(activations:size(1), -1)
    ↪ , 2)
```

Listing 4.4: weights computation

In this section weights variable is computed.

$$\underbrace{weights = torch.sum(\underbrace{gradients : view(\overbrace{activations : size(1)}^{\text{shape} = \ell}, -1)}_{\text{shape} = \ell \text{ x } m^2}, 2)}_{\text{shape} = \ell \text{ x } 1}$$

The weights variable is a $\{\ell x 1\}$ vector. ℓ is the number of filter in the selected layer for the visualisation.

```
-- Summing and rectifying weighted activations across depth
map = torch.sum(torch.cmul(activations, weights:view(activations:
    ↪ size(1), 1, 1):expandAs(activations)), 1)
map = map:cmul(torch.gt(map,0):typeAs(map))
```

Listing 4.5: map computation

Map is finally computed multiplying the weights by the activations. Let's see the code in the detail:

1. $weights : view(activations : size(1), 1, 1) : expandAs(activations)$: reshape weights tensor to shape $\ell \times n \times n$ (all $n \times n$ planes have the same value repeated);

2. $torch.cmul(activations, \cdots, 1)$: activation tensor is elementwise multiplied by the previosly reshaped weights tensor. The result is a weighted activation tensor of shape $\ell \times n \times n$.

3. *torch.sum*$(\cdots, 1)$: all weighted filter from previous step are summed together. Output shape is $1 \times n \times n$
4. The second operation on map tensor is set to 0 all negative map elements.

The final step of the algorithm outcome is a $n \times n$ matrix that used as a colourmap shows the activation regions in the input image.

Chapter 5

Experiments

This chapter explores the results of classification and visualization for all the datasets used.

5.1 Natural images

The visualization techniques are validated on ImageNet trained VGG16. This is a benchmark to test if the visualization is working well before trying on retinal datasets. The network is represented through deep learning libraries as a cascade of tensors.

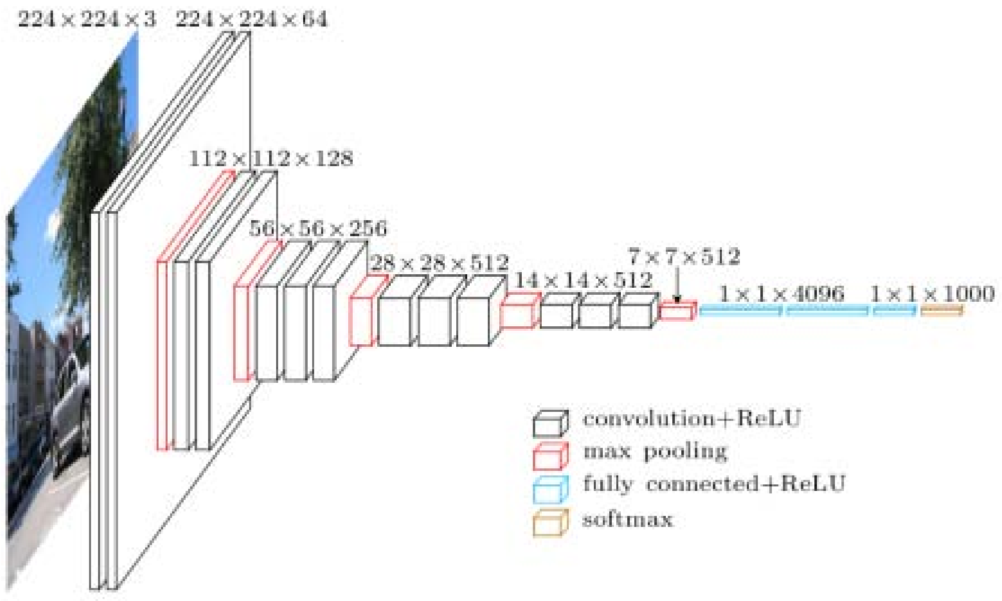


Figure 5.1: VGG16 tensor structure. The input dimension is, for historical reasons, 224x224, with 3 color channels.

The network has been trained on Imagent dataset, with 1 million images divided in 1000 classes.

5.1.1 Filters Visualisation

After training the weights of the convolutional layers are the filter kernels used to perform convolution.

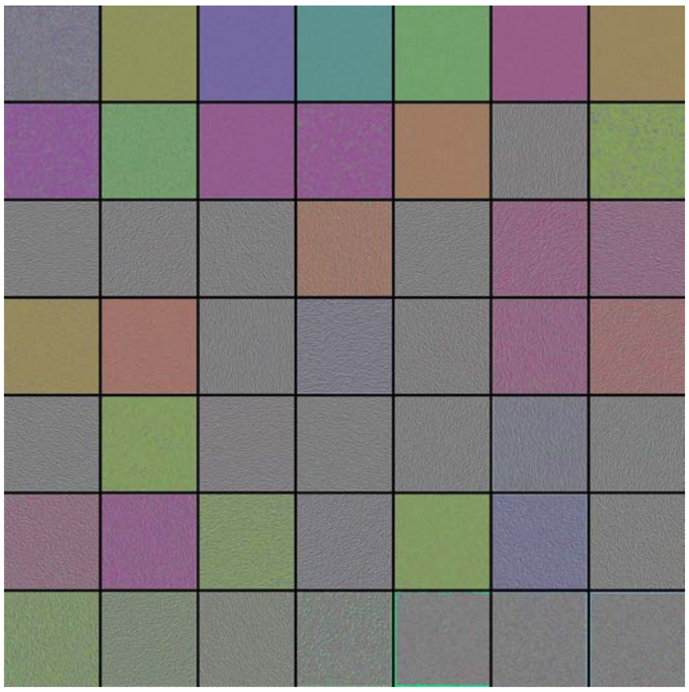


Figure 5.2: Filters of first convolutional layer in block 1.

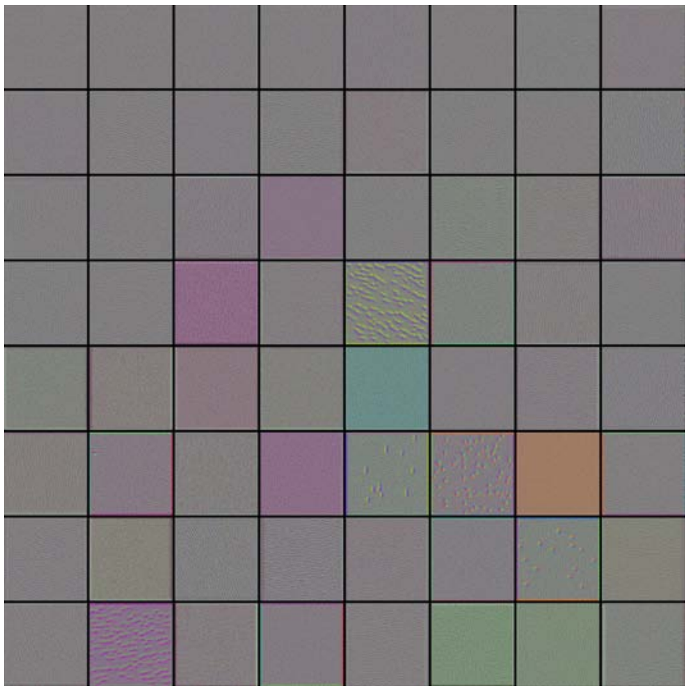


Figure 5.3: Filters of first convolutional layer in block 2.

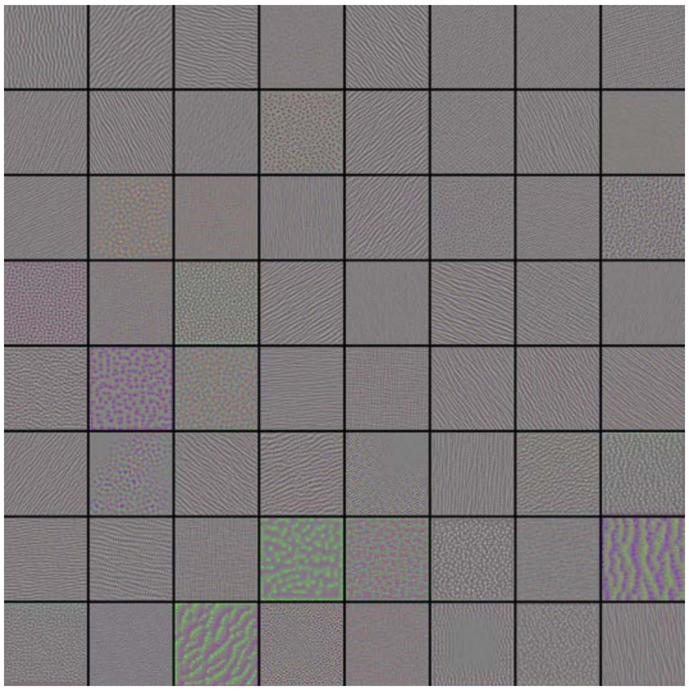


Figure 5.4: Filters of first convolutional layer in block 3.

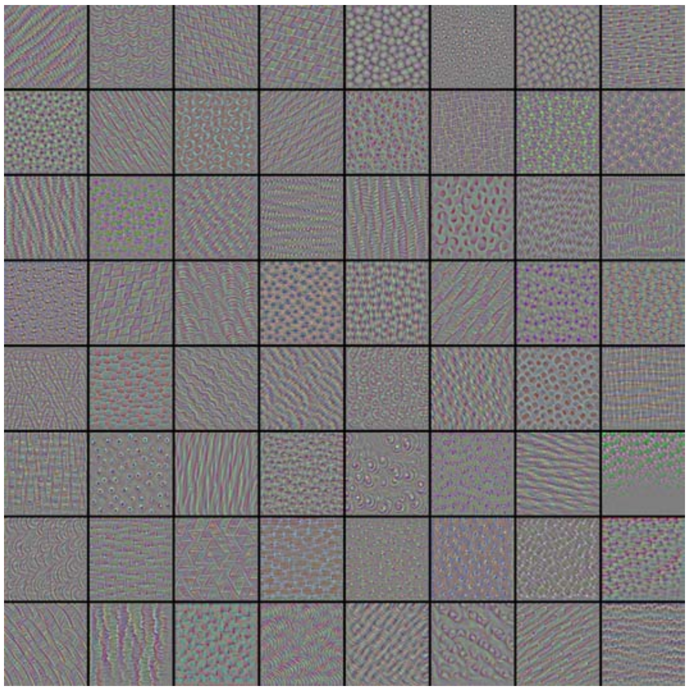


Figure 5.5: Filters of first convolutional layer in block 4.

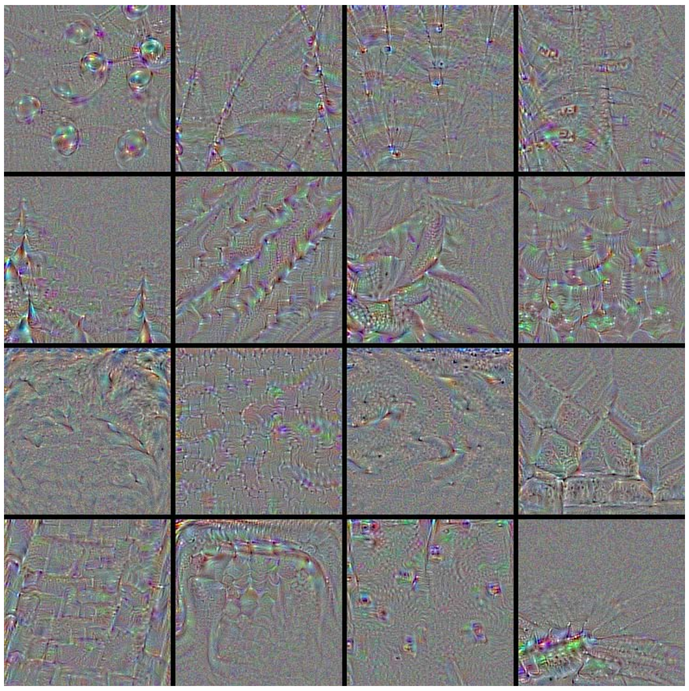


Figure 5.6: Filters of last convolutional layer.

5.1.2 Grad-CAM Visualisation

The Grad-CAM technique performs very well with natural images and ImageNet trained VGG.

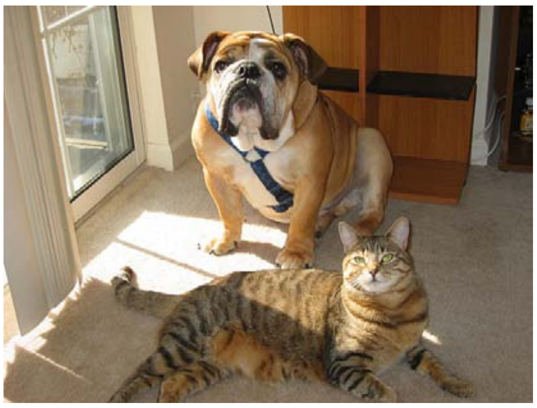


Figure 5.7: Image with a dog and a cat used as input for Grad-CAM test

The previous image is a good example to prove Grad-CAM technique.

Manually setting the class to visualise in the Grad-CAM algorithm, is possible to focus on the cats or dogs class in the same image.

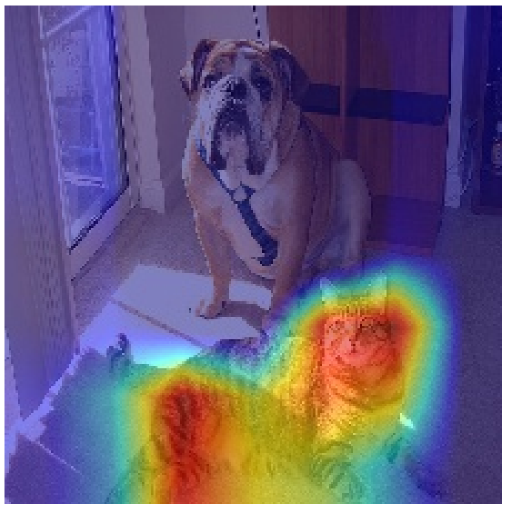


Figure 5.8: Grad-CAM output with class 283 (= Tiger cat) fixed.

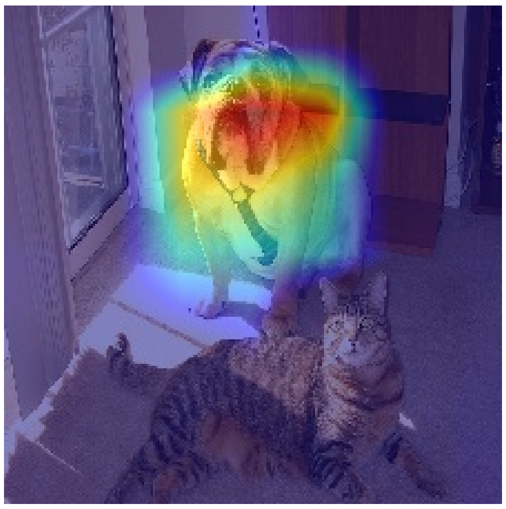


Figure 5.9: Grad-CAM output with class 243 (= Boxer) fixed.

As is possible to see the algorithm outputs a heat-map that shows which part of the image activates the selected class for the selected layer. In this case *Tiger cat* and *Boxer* class has been fixed in two different runs of the program. The output is the activation, firstly, for *Tiger cat* and then *Boxer* classes. Superimposing the heat-map to the original image is possible to see that the highly active regions match with the cat or the dog correctly. This shows that the network is not only classifying correctly, but also using the expected regions in the images.

In the next trials this technique will be applied to medical fundus images on the same architecture trained each time on specific dataset.

5.2 Binary: car , grass blob

5.2.1 Filters visualization

After training the convolutional layers weights are the filter kernels for each stage of the network.

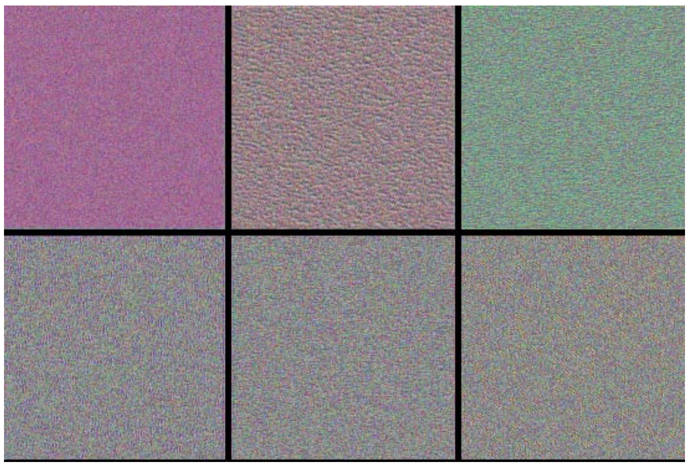


Figure 5.10: Filters of first convolutional layer in block 1.

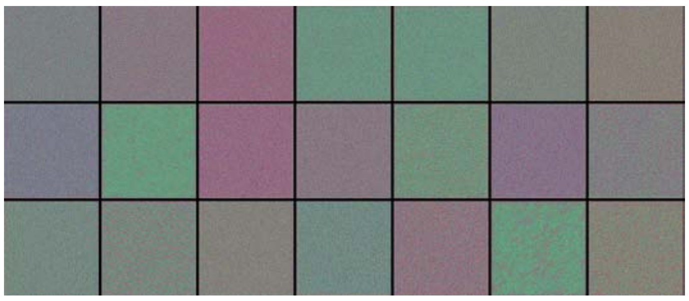


Figure 5.11: Filters of first convolutional layer in block 2.

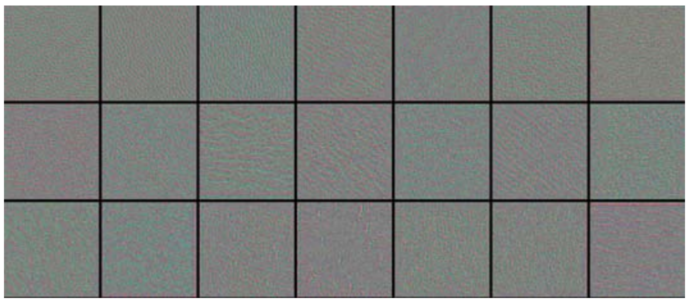


Figure 5.12: Filters of first convolutional layer in block 3.

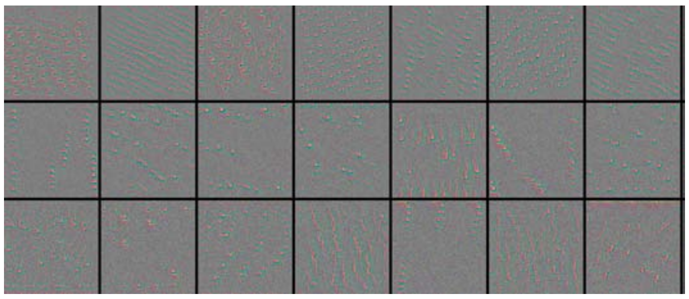


Figure 5.13: Filters of first convolutional layer in block 4.

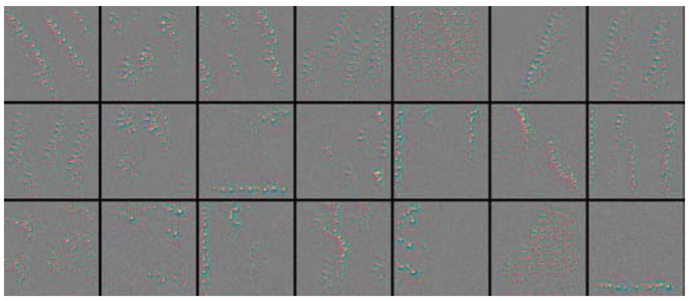


Figure 5.14: Filters of last convolutional layer.

5.2.2 Grad-CAM visualization

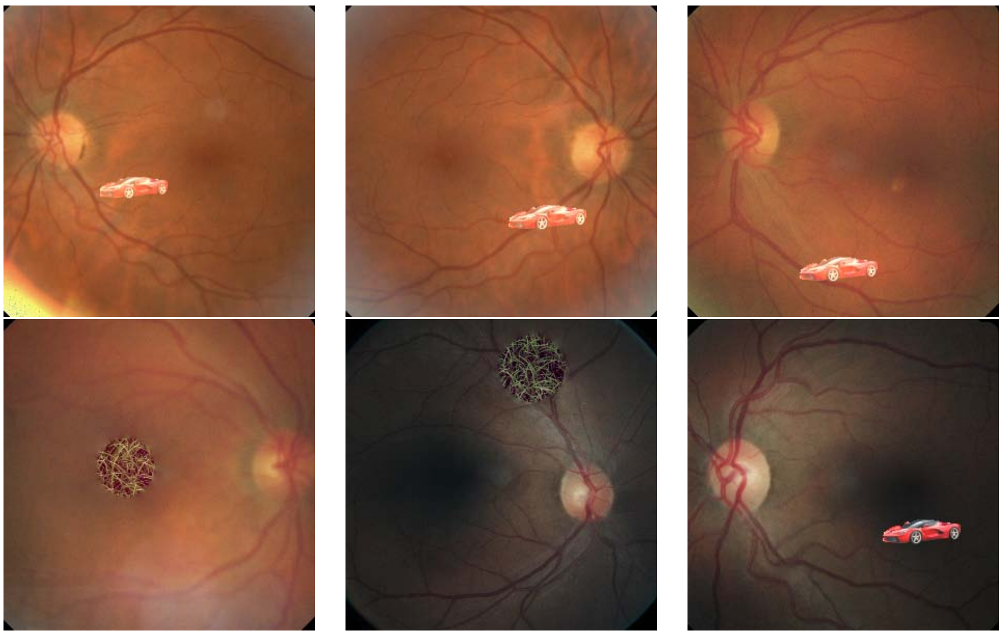


Figure 5.15: Dataset samples. All images contain a car or a grass blob.

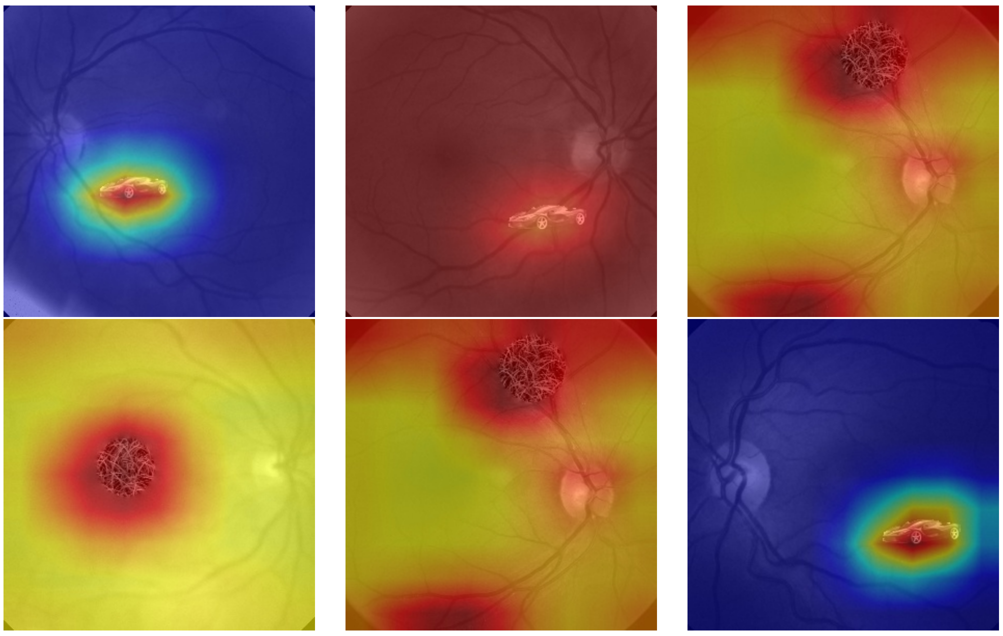


Figure 5.16: Grad-CAM run on the previous samples.

5.3 Multiclass: natural artificial markers

5.3.1 Filters visualization

After training the convolutional layers weights are the filter kernels for each stage of the network.

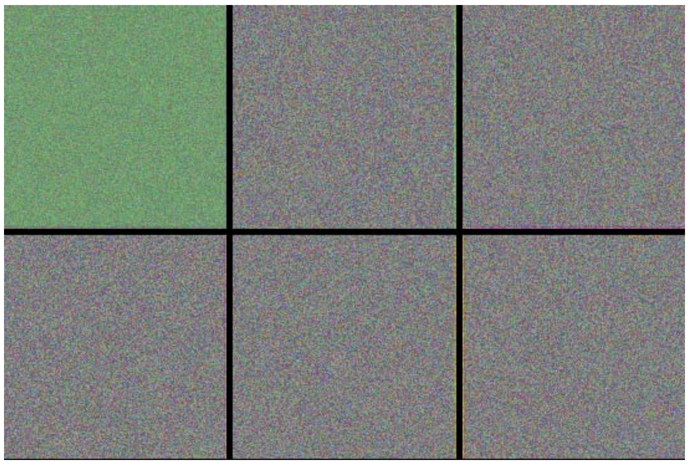


Figure 5.17: Filters of first convolutional layer in block 1.

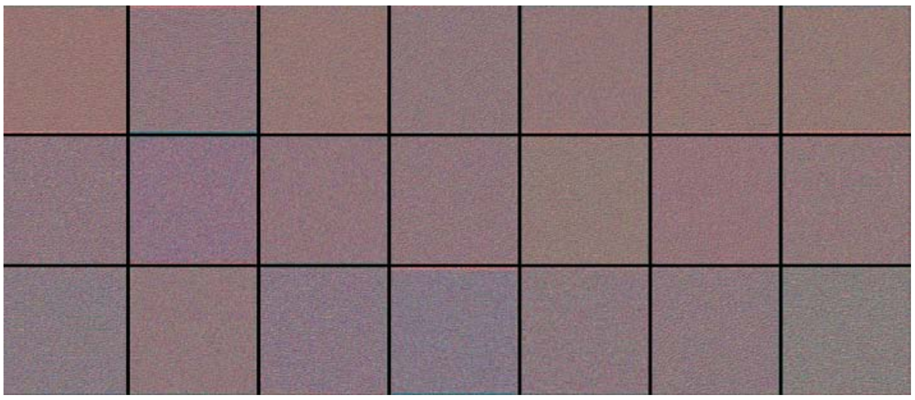


Figure 5.18: Filters of first convolutional layer in block 2.

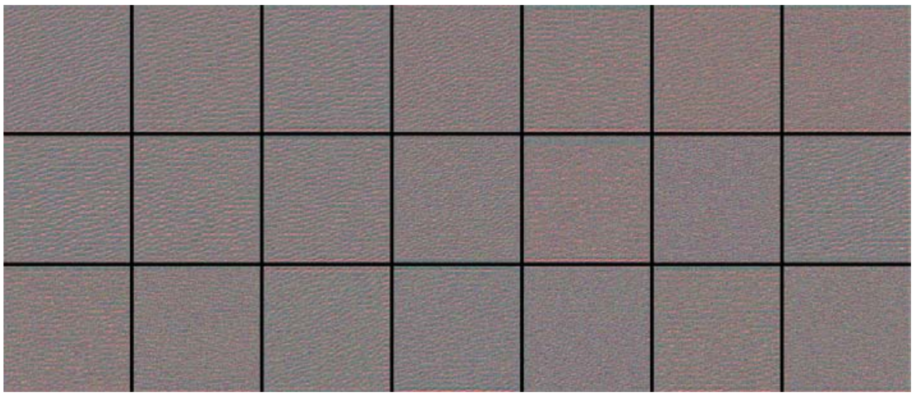


Figure 5.19: Filters of first convolutional layer in block 3.

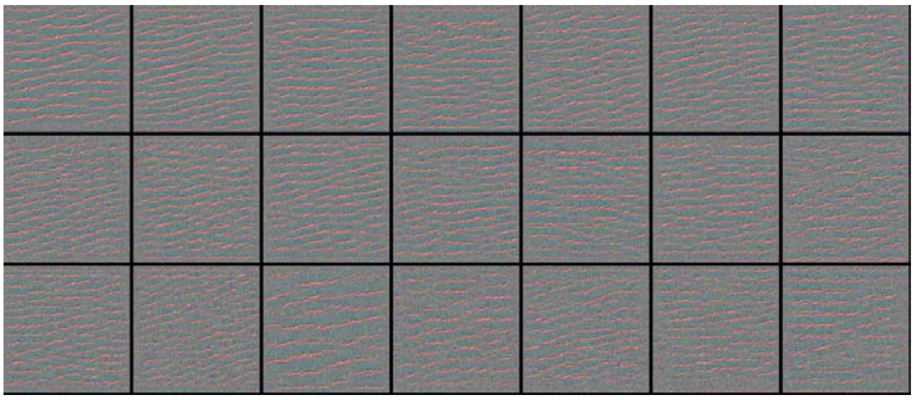


Figure 5.20: Filters of first convolutional layer in block 4.

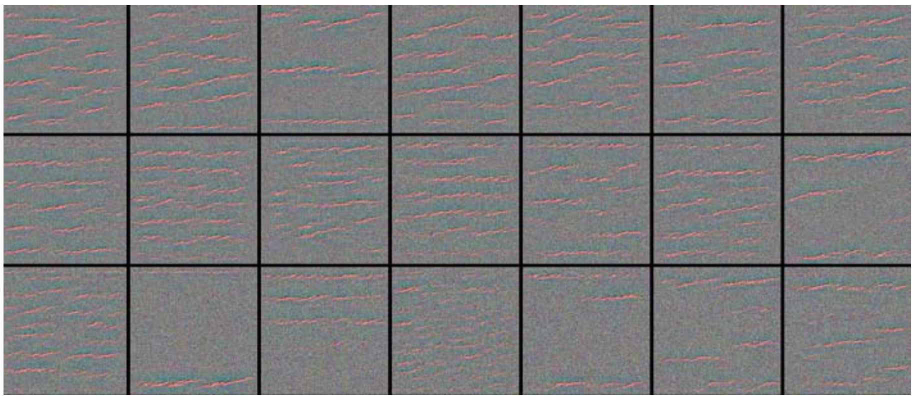


Figure 5.21: Filters of last convolutional layer.

The last layer filters are still very different from Imagenet ones. Moreover there is a significant difference even with the previous that is noticeable in even less complex filters with an horizontal pattern rather repeated among all the filters.

5.3.2 Grad-CAM visualization

The samples used for this dataset show less reliable behave of the Grad-CAM. The marker are still mainly detected by the technique, but in a weaker way than before. Where the visualisation is correct there is more often activation in wrong regions and more importantly in some images the visualisation fails showing an all red heat-map.

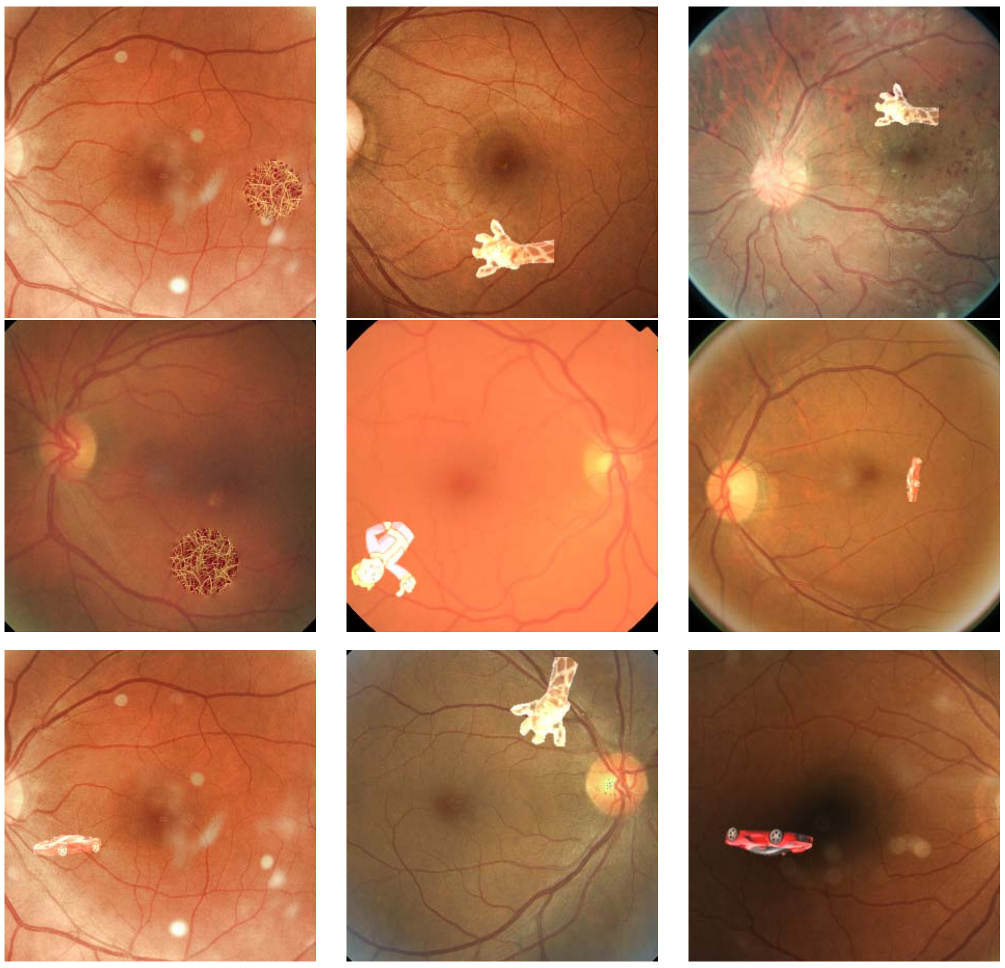


Figure 5.22: Artificial texture markers images samples.

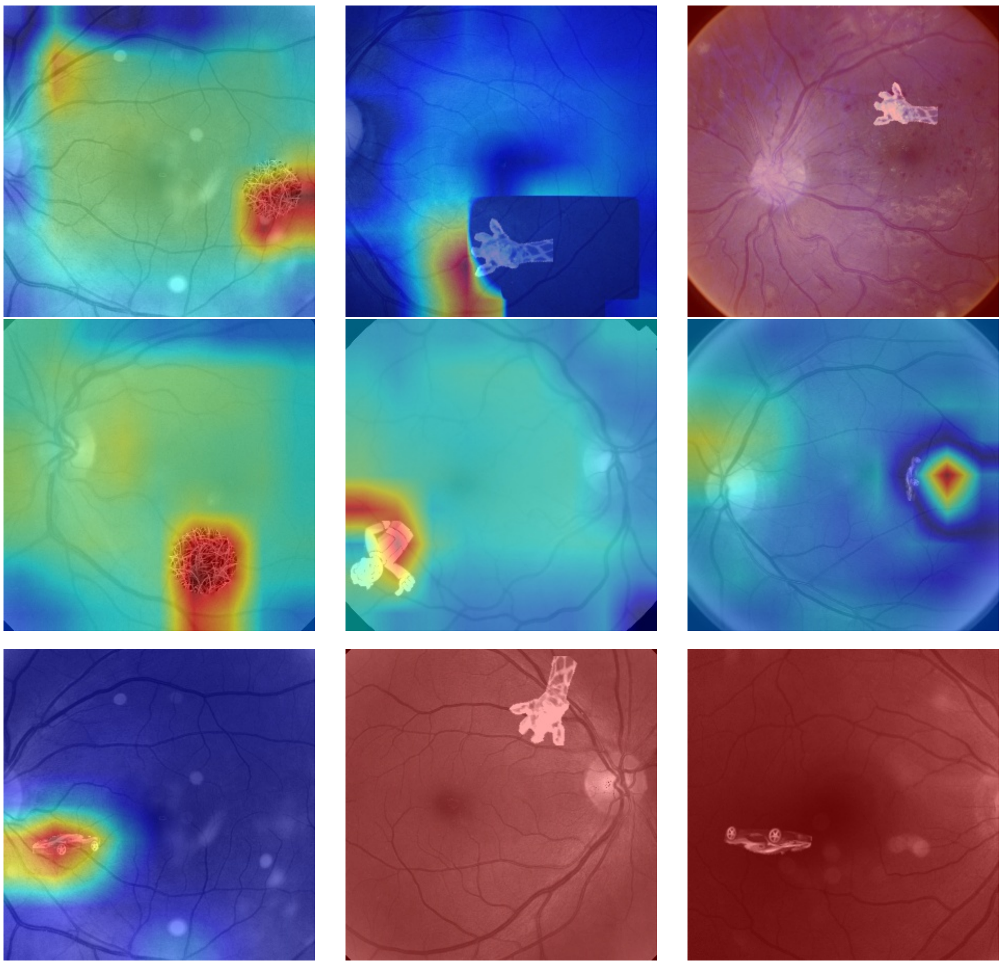


Figure 5.23: Artificial texture markers images samples.

5.4 Binary: grass blob, original fundus images

This dataset is particularly challenging for the visualisation. The classification accuracy is again 100%, but from an intuitive point of view the texture is not important for the classification. The minimum capability needed to discriminate between to fundus images, one with a grass texture blob and one without, is the ability to find circles.

5.4.1 Filters visualisation

The filters for each VGG16 block are printed below.

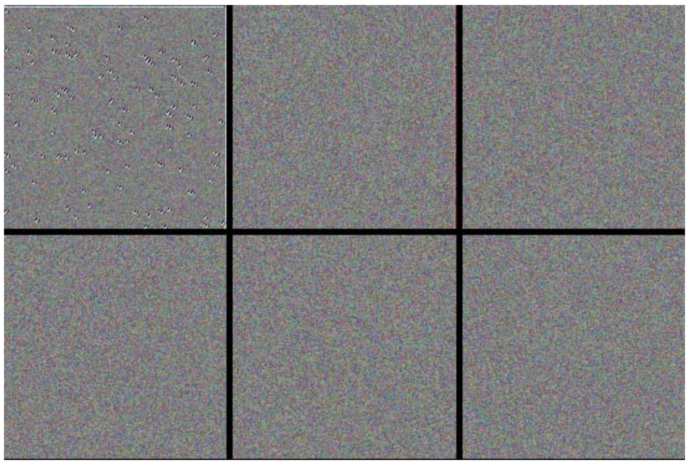


Figure 5.24: Filters of first convolutional layer in block 1.

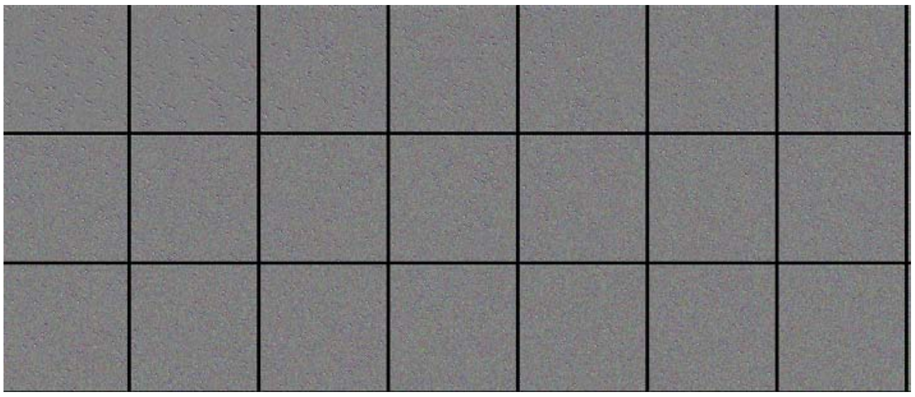


Figure 5.25: Filters of first convolutional layer in block 2.

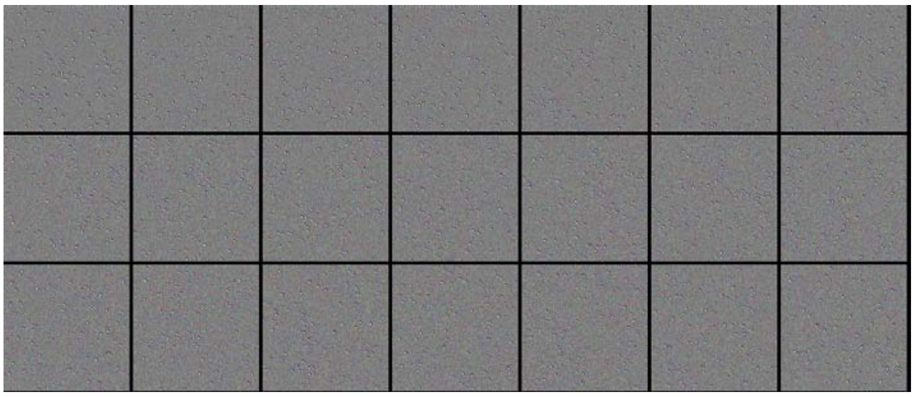


Figure 5.26: Filters of first convolutional layer in block 3.

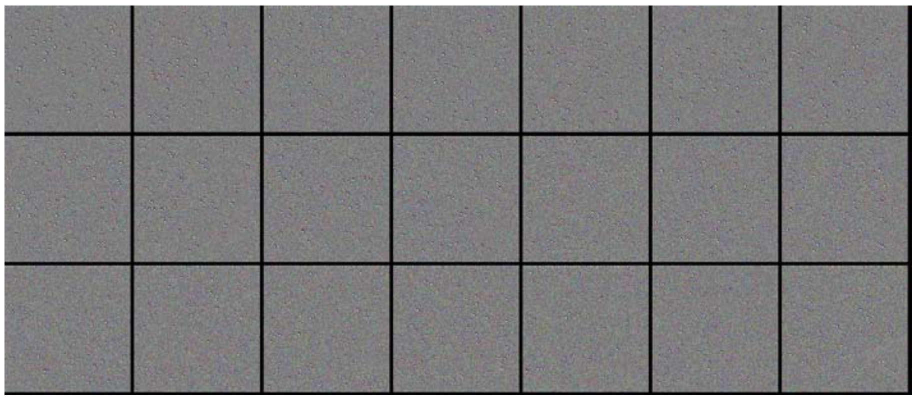


Figure 5.27: Filters of first convolutional layer in block 4.

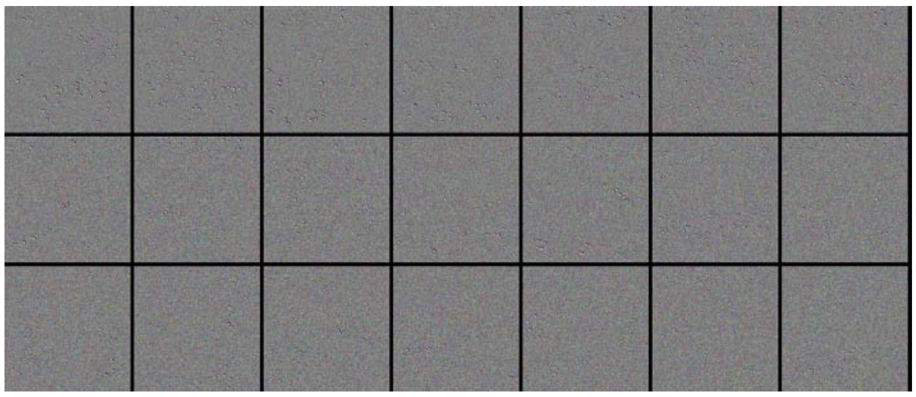


Figure 5.28: Filters of last convolutional layer.

5.4.2 Grad-CAM visualisation

The Grad-CAM result is quite challenging to explain. The visualisation is indeed wrong. The important regions in the images with the grass blob are the part outside the blob. This can be explained recalling the idea explored at the beginning of this dataset section. The network doesn't need to find the blob to classify correctly, but just a circle shape. Probably this is the reason why the Grad-CAM outcome.

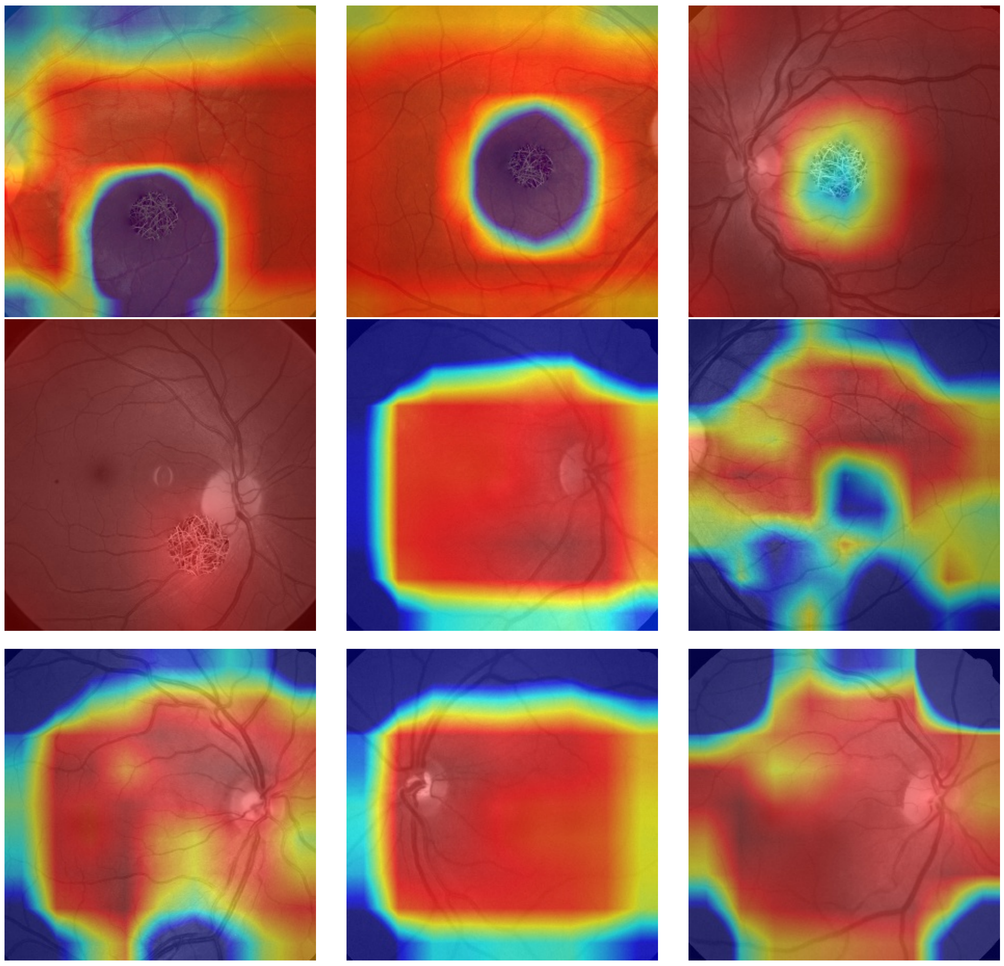


Figure 5.29: Artificial texture markers images samples.

5.5 Texture blobs markers dataset

5.5.1 Filters visualisation

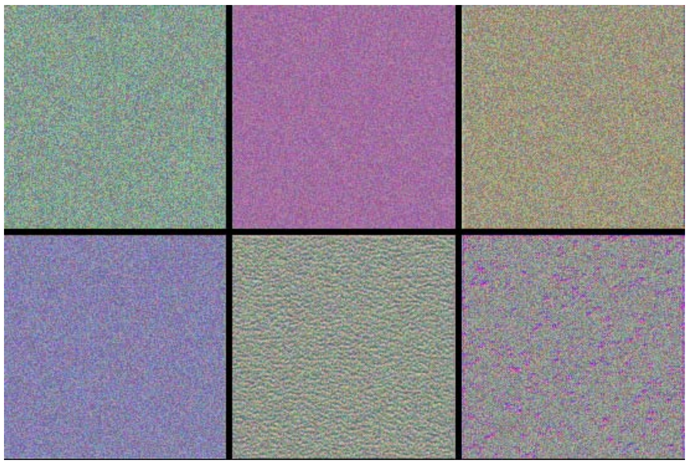


Figure 5.30: Filters of first convolutional layer in block 1.

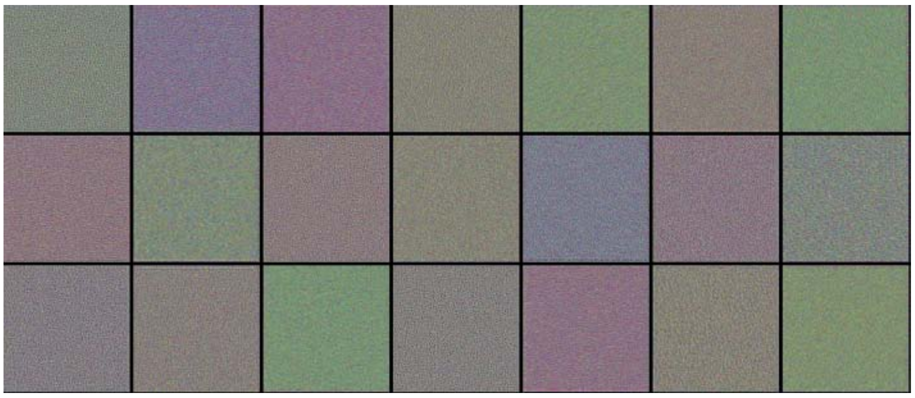


Figure 5.31: Filters of first convolutional layer in block 2.

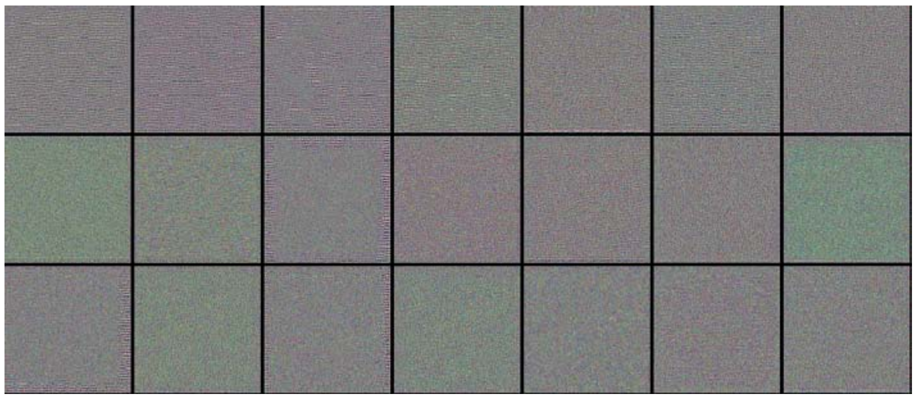


Figure 5.32: Filters of first convolutional layer in block 3.

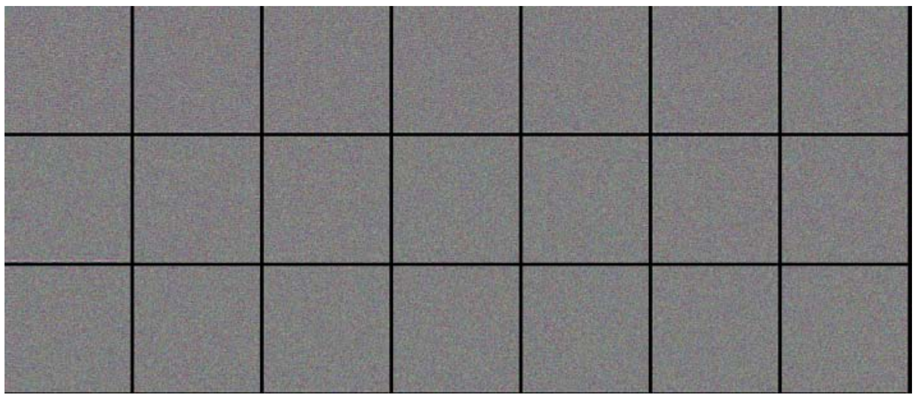


Figure 5.33: Filters of first convolutional layer in block 4.

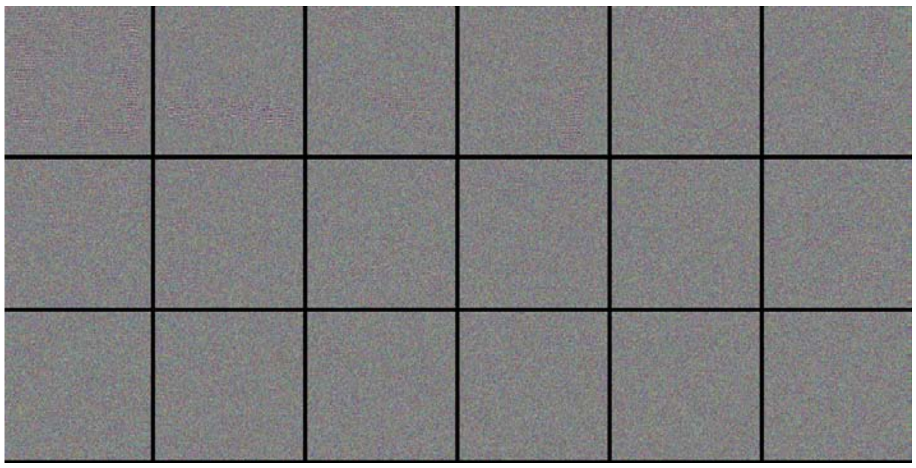


Figure 5.34: Filters of last convolutional layer.

The filters for this database (4 class, one with non modified retina and the other three with different texture blobs) seem to be without a visible pattern. The classification result in validation is still near to 100%. Probably the absence of well visible patterns even in the deeper layers filters is due to the very flat structure of the images; the object to recognise is just texture and probably the fully connected layers at the end are enough to classify without learn anything in the convolutional part of the network.

5.5.2 Grad-CAM visualisation

The previous section suggest that even the Grad-CAM will fail given the shape of the deeper filters.

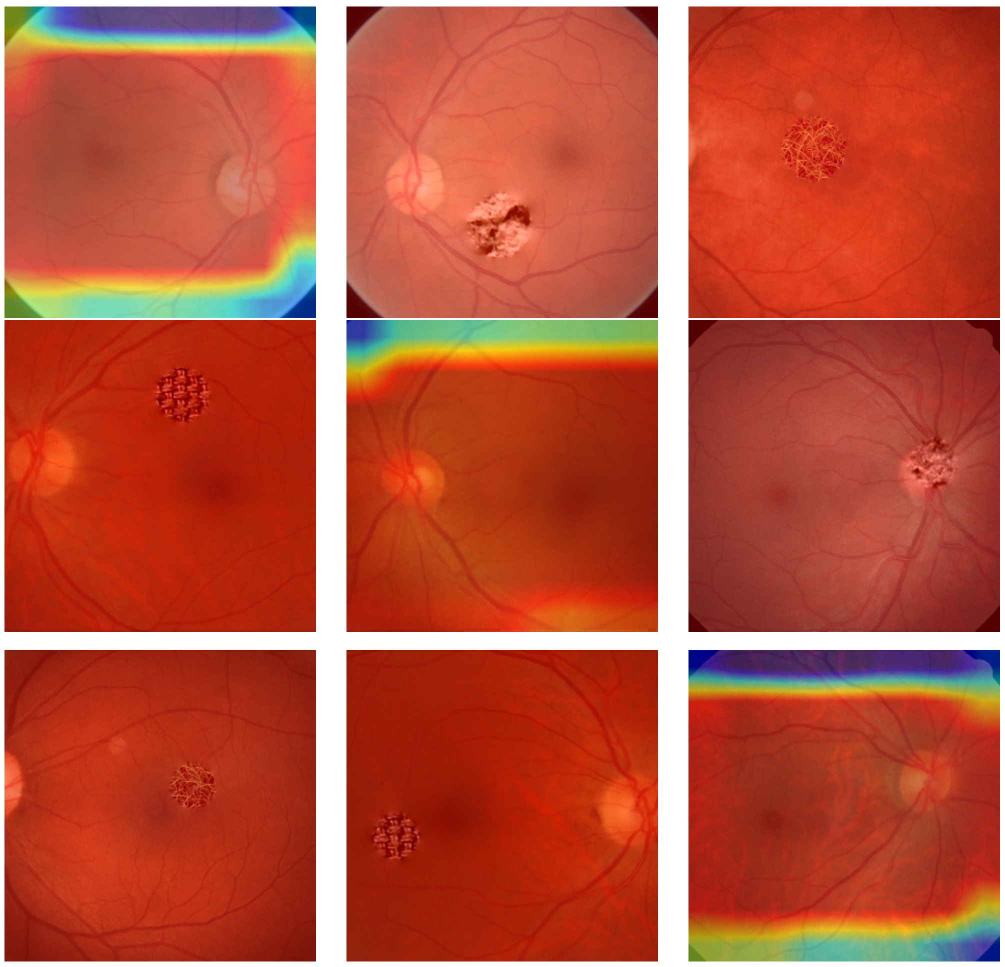


Figure 5.35: Artificial texture markers images visualisation.

The visualisation fails to detect important regions. The classification is 100% in validation, that means that the class can be detected correctly. The limit is in the Grad-CAM algorithm probably because of the filters previously extracted. In the class 0, the one without blobs, is possible to see one or two horizontal bezels with very low activation. The reason is difficult to identify and likely due to absence of any artificial circle inside the image. There is, then, a difference in visualisation between the class without blobs and the others. The result is anyway unsatisfactory and shows a limit of the Grad-CAM with this dataset.

5.6 Diabetic Retinopathy

The classification performance is afflicted by severe overfitting that limits Grad-CAM performances.

5.6.1 Filters visualisation

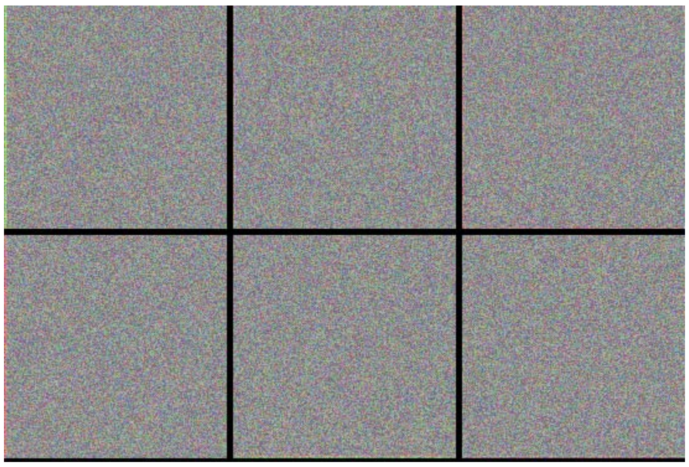


Figure 5.36: Filters of first convolutional layer in block 1.

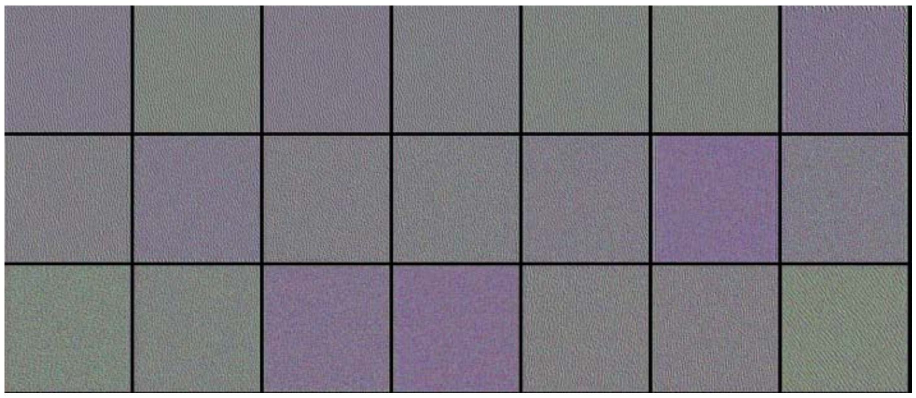


Figure 5.37: Filters of first convolutional layer in block 2.

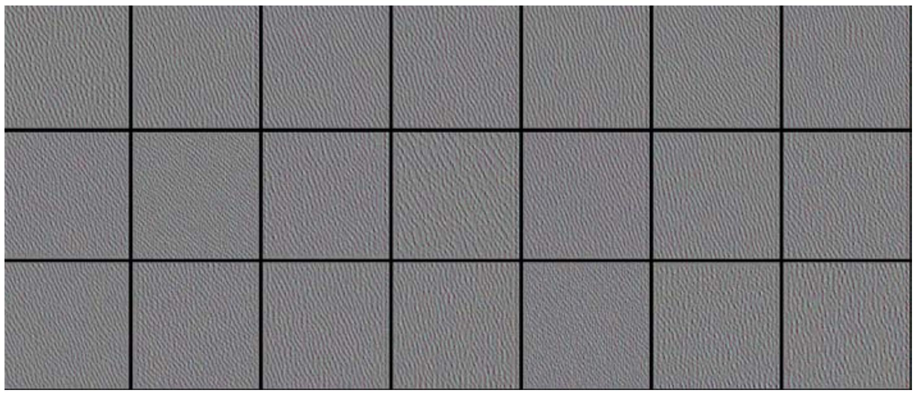


Figure 5.38: Filters of first convolutional layer in block 3.

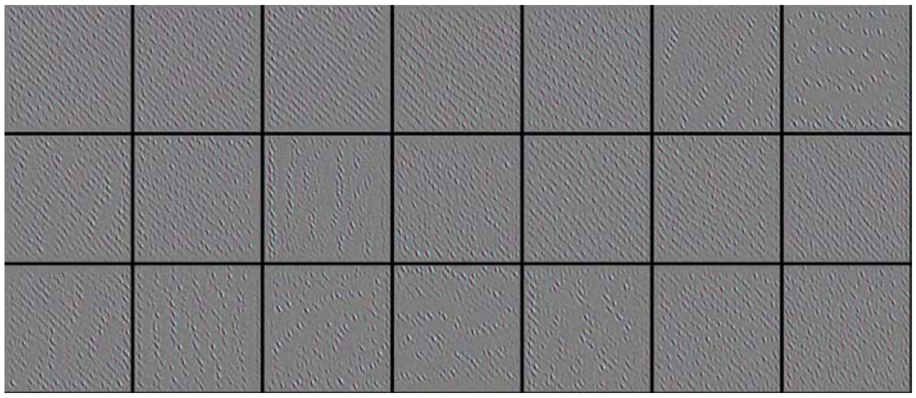


Figure 5.39: Filters of first convolutional layer in block 4.

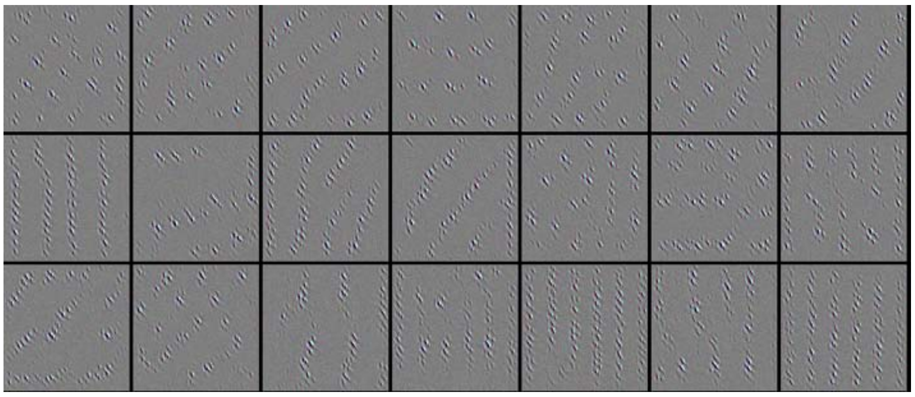


Figure 5.40: Filters of last convolutional layer.

5.6.2 Grad-CAM visualisation

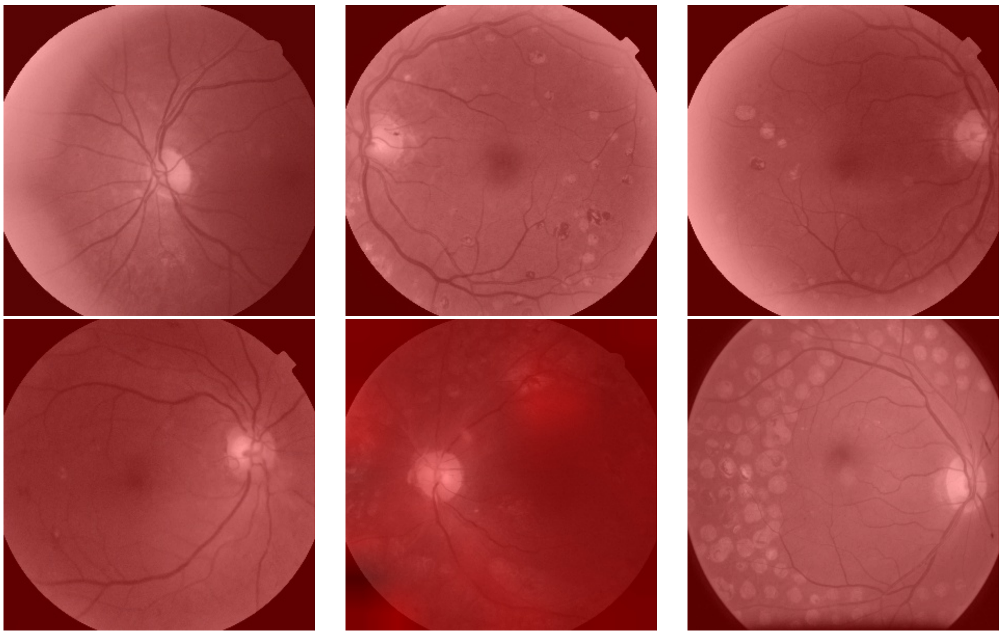


Figure 5.41: Visualisation for diabetic retinopathy.

Chapter 6

Conclusion and Recommendation for future works

The results of this work shows not only the performances and the limits of current the state of the art for visualisation of salient region in Deep Learning, but also shows some interesting point in how a network learns filters.

All the experiments have been run with VGG16 architecture reaching 100% accuracy for artificial markers datasets. Keeping this into account different aspect can be noticed.

6.1 Natural and Medical datasets differences

The first outcome regards the difference between natural images and medical images in Deep Learning. Natural images are plenty of very different objects with very different structures. This allows the network to learn very complex and different filters especially in the deeper layers. Medical images, in particular fundus retinal images, have a lower degree of very different patterns, but very small difference in few details can do the difference in classification. This is testified by the hugh difference in the deeper convolutional layers filters. The ones belonging to a network trained with retinal fundus images are much simpler than the ones obtained from Imagenet dataset. This is also influenced by the few number of different classes in retinal dataset if compared to Imagenet. In the first case the number of classes can vary between 2 and 5, while in the latter is 1000.

6.2 Filters pattern complexity is important

The second outcome regards the limitation of Grad-CAM algorithm performances when the deeper layer filters have simple patterns. The visualisation is less precise and can even fail. Furthermore if the classes are discriminated by natural objects, with standard complex pattern, the performances are good; when the class is discriminated by simple texture blob, with patterns much simple to the retinal background, the visualisation is less strong.

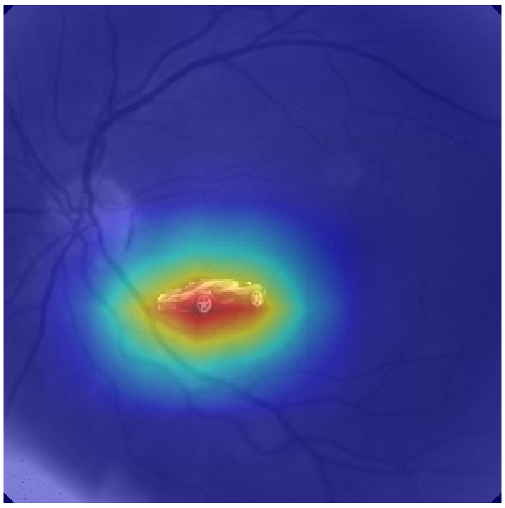


Figure 6.1: Natural image marker: Grad-CAM is usually able to detect with precision the important region.

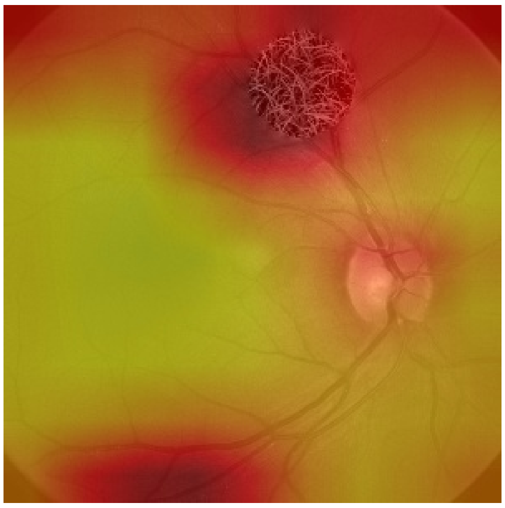


Figure 6.2: Texture blob marker: The discriminator structure is simpler tha the car image in the other class. Moreover the texture is somehow similar to vessels in the retinal background. The result is a less precise visualisation and mild activation in the whole image.

The same comment can be done for the multiclass dataset with natural markers. Here the performances of visualisation are a bit decreased probably because of the filters that despite the higher number of different natural markers are still much similar to the previous dataset than to Imagenet filters.

6.3 Unexpected patterns can be find

The third outcome regards the importance of the kind of classes to be discriminated. If the user want to classify between to classes, the first with a pattern, the latter without, the visualisation performances can be very different from what expected. In the binary dataset tested where the classes are one with a grass blob and one without, the user can expect that Grad-CAM can easily find the blob in the respective class. However, this is not what happens in the dataset tested.

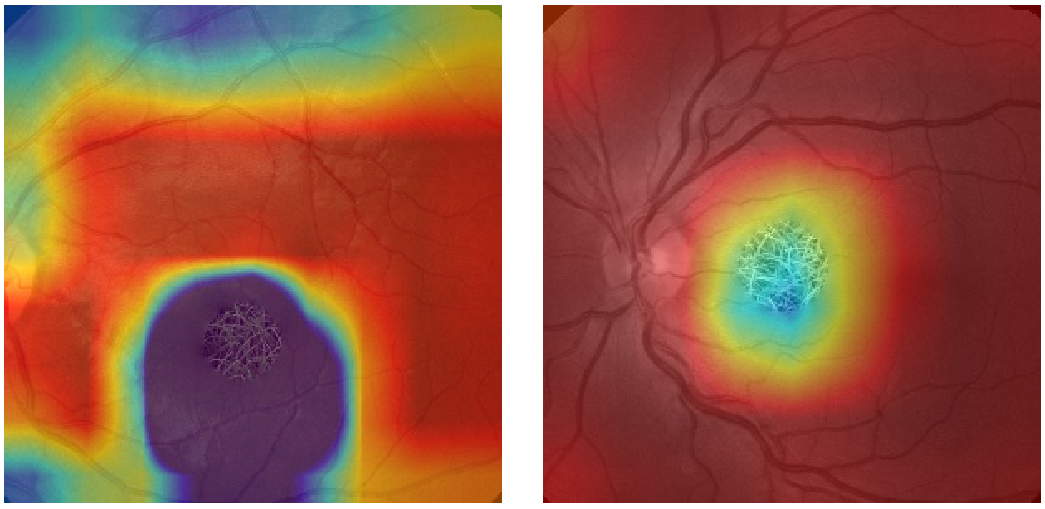


Figure 6.3: Grass blob visualisation outcome

The visualisation outcome are somehow weird and difficult to explain. Indeed, in Figure 6.3, the visualisation suggest that the discriminative region for the blob class is the background and not the blob itself. A possible interpretation to this is that in this case is not mandatory to find the blob with its texture; a circle pattern is enough to discriminate between class with or without this pattern. This can be confirmed by the outcome of the class without the blob.

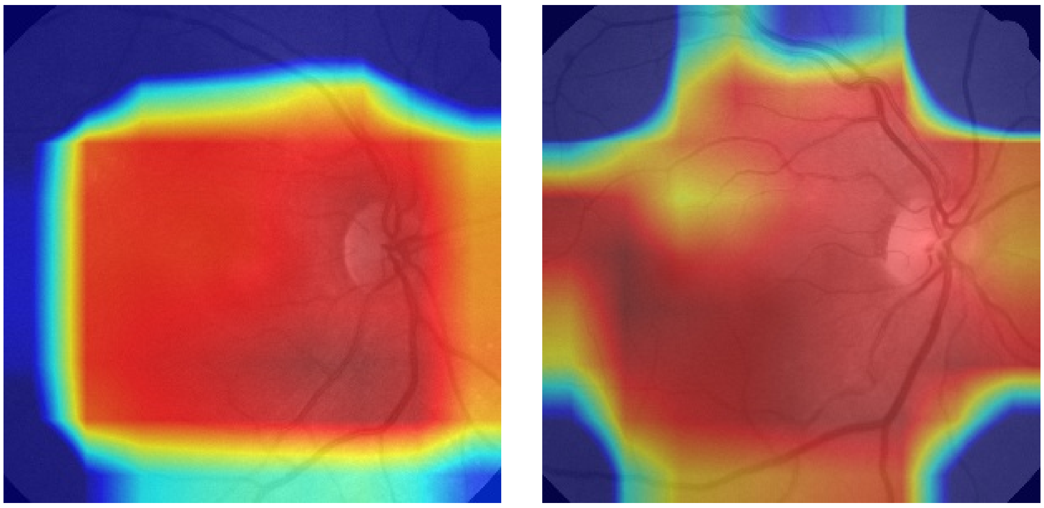


Figure 6.4: Visualisation outcome for *no blob class*

In Figure 6.4 shows how the activation region is approximately a square in the center of the image. The classification is possibly done filtering circles in the image. If the image does not contain circle all the central square is active and the image is classified as *no blob* class. If not all the central part of the image activates the discriminative filters then a circle is present and the blob detected. In the last layer filters seems not to be present a single filter able to find circle, but this can easily be a combination between the 512 last layer filters.

This show that sometimes the network is able to discriminate different classes using not expected patterns. This is a very interesting point and shows that a network is able to discover not expected patterns and visualisation can be a preliminary proof.

6.4 Filters pattern can limit the performances

Texture multiclass dataset visualisation fails. This is probably due to the filters. The last layer filters does not contain any strong visible pattern. Grad-CAM algorithm is indeed very dependent on the filters. Again the class with no blob seems to be find activating the central square of the image. Unfortunately few can be said about the other classes, given that there is no sign at all of activation with 100% accuracy performance.

6.5 Diabetic Retinopathy visualisation

Visualisation fails for Diabetic Retinopathy dataset. This is probably not due to Grad-CAM but to limited classification performances. The classification gives, indeed, strong overfitting limiting the validation accuracy.

- Trianing set accuracy: 99%;
- Validation set accuracy: 70%

The limit seems due to this. Improving the classification performances reducing the overfitting is fundamental to get the visualisation working. This is a very tough task that is not part of the aim of this thesis and can be part of a future work.

Appendices

Appendix 1

Jacobian of composite functions:

$$g : \mathbb{R} \to \mathbb{R}^n$$
$$g : \mathbb{R}^n \to \mathbb{R}$$

defining:

$$h = f \circ g : \mathbb{R} \to \mathbb{R}$$

is possible to write:

$$\begin{aligned} h'(\mu) &= (J_f)(\boldsymbol{x})^0 (J_g)(\boldsymbol{\mu}^0) = \\ &= \left[\frac{\delta f}{\delta x_1}(\boldsymbol{x}^0), \cdots, \frac{\delta f}{\delta x_n}(\boldsymbol{x}^0) \right] \cdot \begin{bmatrix} g_1'(\mu_0) \\ \vdots \\ g_n'(\mu_0) \end{bmatrix} = \\ &= \left\langle \nabla f(\boldsymbol{x}^0), g'(\mu_0) \right\rangle \end{aligned} \tag{1}$$

Appendix 2

Preprocessing python code:

```
#Preprocess training images.
import cv2, numpy, multiprocessing
from os import listdir
from os.path import isfile, join

def scaleRadius(img,q,rd):
    x=img[img.shape[0]/2,:,:].sum(1)
    r=(x>x.mean()/10).sum()/2
    if img.shape[0] >= r*2*q:
        cimg = img[img.shape[0] / 2 - int(r*q):img.shape[0] / 2 +
            ↪  int(r*q), img.shape[1] / 2 - int(r*q):img.shape
            ↪ [1] / 2 + int(r*q), :]
    else:
        cimg = img[:, img.shape[1] / 2 - int(r*q):img.shape[1] /
            ↪ 2 + int(r*q), :]
    return (cimg,r)

def processing(f):
    try:
        a = cv2.imread('C:/Users/enricovincenzi/KAGGLE_DATASET/
            ↪ TRAIN/train1/'+ f)
        q = 0.93
        r = 1150
        [a, r1] =scaleRadius(a, q, r)
        b=numpy.zeros(a.shape)
        cv2.circle(b,(a.shape[1]/2,a.shape[0]/2),int(r1*q)
            ↪ ,(1,1,1),-1,8,0) #int(scale*0.9)
        aa = cv2.addWeighted(a, 4, cv2.GaussianBlur(a, (0, 0),
            ↪ 33), -4, 128) * b + 128 * (1 - b)
        cv2.imwrite('C:/Users/enricovincenzi/KAGGLE_DATASET/TRAIN
            ↪ /train2/' + f, a)
        print(f)
    except:
        print f

if __name__ == '__main__':
    mypath = 'C:/Users/enricovincenzi/KAGGLE_DATASET/TRAIN/train1
        ↪ '
    images = [f for f in listdir(mypath) if isfile(join(mypath, f
        ↪ ))]
    pool = multiprocessing.Pool()
    pool.map(processing, images)
    pool.close()
    pool.join()
```

Listing 1: Preprocessing and bezels cut.

Appendix 3

Artificial markers image generator code:

```
size = 300                    # final size of each image
crop = 240                    # reduce the radius of the image of
    ↪ crop pixels (introduces cuts)
fake_biom = 1                 # 1 for draw texture in the images, 0
    ↪  to keep the image as it is

###### basic functions ######

def square(image):
    mshape = max(image.shape[:2])
    square = np.zeros([mshape, mshape, 3], dtype='uint8')
    shape = image.shape[:2]
    middle, pos = [np.min(shape)/2, np.argmin(shape)]
    f = 0 if np.min(shape) % 2 == 0 else 1
    # !! 4 times slower
    # square[mshape / 2 - middle:mshape / 2 + middle + f, :, :] =
        ↪  image if pos == 0 else image.swapaxes(0,1)
    if pos == 0:
        square[mshape / 2 − middle:mshape / 2 + middle + f, :, :]
            ↪  = image
    if pos == 1:
        square[:, mshape / 2 − middle:mshape / 2 + middle + f, :]
            ↪  = image
    return square


def resize(image):
    dsize = randint(40,100)
    #print(dsize)
    resized = cv2.resize(image, (dsize, dsize))
    return resized


def rotate(image):
    (h, w) = image.shape[:2]
    center = (w / 2, h / 2)
    M = cv2.getRotationMatrix2D(center, randint(0, 360), 1.0)
    rotated = cv2.warpAffine(image, M, (w, h))
    return rotated

################################

mypath = 'path for the images'
files = [f for f in listdir(mypath) if isfile(join(mypath, f))]
files = sorted(files, key=lambda x: (int(re.sub('\D','',x)),x))
len = files.__len__()

f = h5py.File('kaggle_' + str(size) + '_boyycarr.hdf5', 'a')
data = f.create_dataset("dataset", (len, size, size, 3), dtype='
   ↪ uint8', chunks=True)
label = f.create_dataset("label", (len, 1), dtype='uint8', chunks
   ↪ =True)
g = h5py.File('texture.hdf5', 'r')
texture = g.get('dataset')
car = square(cv2.imread('PNG/ferrari100.png'))
gir = square(cv2.imread('PNG/giraffe100.png'))
boy = square(cv2.imread('PNG/vaultboy100.png'))
droid = square(cv2.imread('PNG/droid100.png'))
# fake_biom creator

def fake_biomarkers(image):
    tran = np.zeros([size, size, 3], dtype='uint8')
    fbiom = [car, boy] #['blob', car, gir, boy, droid]
    lb = random.randint(0, fbiom.__len__()-1)
    a = fbiom[lb]
    if isinstance(a, str):
        sr = np.zeros((size, size, 3), dtype=np.uint8)
        r = randint(round(size / 12), round(size / 8))  # radius
            ↪ of the biom
        c1 = randint(0 + r, size - r)  # first center coordinate
        h = float(abs(c1 - size / 2)) / float((size / 2 - r))
        h1 = round(math.sin(math.acos(h)) * (size / 2 - r))
        c2 = randint(size / 2 - h1, size / 2 + h1)
        center = (c1, c2)
        mask = cv2.circle(sr, (c1, c2), r, (255, 255, 255), -1)
        typ = 0 # typ = randint(0, 3)
        src = texture[typ, :, :, :]
        image = cv2.seamlessClone(src, image, mask, center, cv2.
            ↪ NORMAL_CLONE)
        return [image, lb]
    d1 = randint(0, size - a.shape[0])
    d2 = randint(0, size - a.shape[1])
    a = resize(a)
    a = rotate(a)
    tran[d1 : d1 + a.shape[0], d2 : d2 + a.shape[1], :] = a
    image = cv2.addWeighted(image, 1, tran, 1, 0)
    return [image, lb]


i = 0
for myFile in files:
    print myFile
    image = cv2.imread(mypath + myFile, -1)
    chk = image.shape[1]
    due = image.shape[0]
    if chk < size + crop or due < size + crop:
        continue
    if size < chk:
        if chk - due < 5:
            chk = chk / 2
            image = image[chk - chk + crop:chk + chk - crop, chk
                ↪ - chk + crop:chk + chk - crop, :]
            image = cv2.resize(image, (size, size))
            if fake_biom == 1:
                [image, a] = fake_biomarkers(image)
            data[i, :, :, :] = image
            label[i] = a
            i = i + 1
        else:
            due = due / 2
            chk = chk / 2
            if due - chk + crop > 0:
                image = image[due - chk + crop:due + chk - crop,
                    ↪ chk - chk + crop:chk + chk - crop, :]
                image = cv2.resize(image, (size, size))
                if fake_biom == 1:
                    [image, a] = fake_biomarkers(image)
                data[i, :, :, :] = image
                label[i] = a
                i = i + 1


data.resize((i, size, size, 3), None)
label.resize((i, 1), None)
f.close()
```

Listing 2: Add artificial markers to fundus images.

Bibliography

[1] G. V, P. L, C. M, and et al, "Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs," *JAMA*, vol. 316, no. 22, pp. 2402–2410, 2016.

[2] D. E. Worrall, C. M. Wilson, and G. J. Brostow, *Automated Retinopathy of Prematurity Case Detection with Convolutional Neural Networks*, pp. 68–76. Cham: Springer International Publishing, 2016.

[3] J. Shan and L. Li, "A deep learning method for microaneurysm detection in fundus images," in *2016 IEEE First International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*, pp. 357–358, June 2016.

[4] H. Pratt, F. Coenen, D. M. Broadbent, S. P. Harding, and Y. Zheng, "Convolutional neural networks for diabetic retinopathy," *Procedia Computer Science*, vol. 90, pp. 200 – 205, 2016.

[5] T. Schlegl, S. M. Waldstein, W.-D. Vogl, U. Schmidt-Erfurth, and G. Langs, "Predicting semantic descriptions from medical images with convolutional neural networks," in *IPMI*, 2015.

[6] K. Maninis, J. Pont-Tuset, P. A. Arbeláez, and L. J. V. Gool, "Deep retinal image understanding," *CoRR*, vol. abs/1609.01103, 2016.

[7] R. Srivastava, J. Cheng, D. W. K. Wong, and J. Liu, "Using deep learning for robustness to parapapillary atrophy in optic disc segmentation," in *2015 IEEE 12th International Symposium on Biomedical Imaging (ISBI)*, pp. 768–771, April 2015.

[8] P. Prentašić and S. Lončarić, "Detection of exudates in fundus photographs using convolutional neural networks," in *2015 9th International Symposium on Image and Signal Processing and Analysis (ISPA)*, pp. 188–192, Sept 2015.

[9] T. Mitchell, *Machine Learning.* McGraw-Hill, 1997.

[10] http://web.media.mit.edu/~lieber/PBE/what-is-PBE.html. Accessed: 2017-03-22.

[11] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2009.

[12] https://online.science.psu.edu/bisc004_activewd001/node/1907. Accessed: 2017-03-22.

[13] http://cs231n.github.io/convolutional-networks. Accessed: 2017-03-22.

[14] https://keras.io. Accessed: 2017-04-02.

[15] F. Chollet, "keras." https://github.com/fchollet/keras, 2015.

[16] https://www.tensorflow.org. Accessed: 2017-04-02.

[17] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, (GA), pp. 265–283, USENIX Association, 2016.

[18] Y. Tang, "Tf.learn: Tensorflow's high-level module for distributed machine learning," *CoRR*, vol. abs/1612.04251, 2016.

[19] http://torch.ch. Accessed: 2017-04-02.

[20] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.

[21] G. van Rossum and F. D. (eds), "Python reference manual," in *http://www.python.org*, 2001.

[22] https://www.lua.org/. Accessed: 2017-04-02.

[23] http://luajit.org/. Accessed: 2017-04-02.

[24] http://www.nvidia.com/object/cuda_home_new.html. Accessed: 2017-04-02.

[25] `https://www.kaggle.com/c/diabetic-retinopathy-detection`. Accessed: 2017-04-02.

[26] `http://www.eyepacs.com`. Accessed: 2017-04-02.

[27] `https://www.jetbrains.com/pycharm/`. Accessed: 2017-04-02.

[28] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[30] R. R. Selvaraju, A. Das, R. Vedantam, M. Cogswell, D. Parikh, and D. Batra, "Grad-cam: Why did you say that? visual explanations from deep networks via gradient-based localization," *CoRR*, vol. abs/1610.02391, 2016.

[31] B. Zhou, A. Khosla, L. A., A. Oliva, and A. Torralba, "Learning Deep Features for Discriminative Localization.," *CVPR*, 2016.