PlantVillage Bachelor 2017

Ioannis Noukakis

Under the supervison of:

Sharada Prasanna Mohanty, Perez-Uribe Andres and Marcel Salathé

1 Specifications

PlantVillage is an open access database of 50,000+ images of healthy and diseased crops. The objective is to enable the development of open access machine learning algorithms that can accurately classify crop diseases on a smartphone. PlantVillage is built on the premise that all knowledge that helps people grow food should be openly accessible to anyone on the planet. Within the framework of this project in collaboration with the Laboratory of Digital Epidemiology of Prof. Salathé from EPFL, we will apply different feature extraction techniques to the images of the mentioned database and will use Machine Learning techniques to build classifiers capable of discriminating between healthy and diseased crops or capable of identifying the disease. The student will take advantage of GPU-accelerated Machine Learning and Image Processing libraries in Python (e.g., OpenCV, Keras) and will provide thorough analyses of his experiments.

Contents

2 Summary

In this Bachelor work, we first learned the tools and libraries that were required in order to do this project. We choosed to use Python because of Keras: a high level framework for machine learning that relies on Tensorflow for computational operations. Tensorflow is a distributed and GPU accelerated computational framework. As we did this task, we managed to make a fully operational environment for large data processing with machine learning. As stated in the specifications, we had to take advantage of GPU acceleration for machine learning and so we did.

Once we were comfortable with these tools, we decided to focus on a specific question that is "Is there a way to quantify bias in a data set". By bias, we mean anything that could lead our self learning tool to weaker results. So we took the PlantVillage data set "as it was" and started investigating. One immediate bias, we found, was the color difference between classes. If every class has different lighting conditions the self learning tool could just learn these lighting differences and gives completely wrong predictions on real life application. This problem was already corrected in our data set.

Then the question of the background arose. Can a certain type of background cause bias in the learning phase of our self learning algorithm ? The bias here would be that our self learning tool would use the background to do its classification instead of the leaves. To answer this question we fine-tuned a model called VGG16 with four types of background data set (untouched background, abstract shapes background, randomly generated background and black background) and we used a process called Class Activation Mapping to see if any types of background was indeed inducing bias to our model. Pushing the thinking further, we also developed a bias metric about how much is our self learning tool is paying attention to the background ratio. In this Bachelor work, we learnt that the background plays a major role in the training of a convolutional neural network.

3 Introduction

3.1 Frame

Today, modern technology allows to grow vegetables in enough quantities to supply food to billions of people. However, diseases remain a major threat to these food resources and a large part of the crops is lost each year. And its extremely difficult, for a small farmer who counts on his subsistence production to get in contact with an expert that could deliver a diagnostic on his diseased crops. In Africa alone, 80% of the agricultural output comes from smallholder farmers.

With billions of smartphone around the globe, why not use them as a diagnostic tool? One could use the images from the camera and get a diagnosis of the disease based on the aspect of the plant.

This is how PlantVillage[1] was born. It is a database of images plants and their known diseases. Its also a platform where a community of experts in agriculture provides help and information to those wishing to grow food.

Machine learning has been for the past few years on the hype train and nowadays powerful libraries and cheap hardware provide easy access to it. For instance, we can train a neural network in just 6 lines of Python with the Keras[2] library! So in the past few years, a large community of scientists and engineers was born. It is gathering and constantly improving as testing machine learning, with new experiments leading to breakthroughs.

3.2 Statement of purpose

The motivation for this Bachelor work was first to explore the self learning tools that exist and train one in order to automate disease classification process. As we explored thoses tools, the question of bias arose. We knew that our self learning tool was reaching very good performances, but we had no idea which features on the images our tool used to classify the images. Some observations were done before for such bias and we even managed to make our own. In order to answer those questions, several experiments were done by tweaking the image's background and a tool to quantify bias in a data set has been developed.

4 State of the art

Simples tasks such as facial recognition or object detection are dead simple to humans, but nearly impossible for computers. This is because computers are programmed to follow a defined set of rules. Ideally, they have to learn the rules that we, humans, cannot program. To do so we developed rules on how to learn rules. This paradigm is called machine learning. Inspired by the central nervous system of humans, Artificial Neural Networks have been developed. Like their biological counterpart they are built upon signal processing and are connected together just like in the human brain.

Nowadays there are tree main paradigms to machine learning:

- Supervised learning: We use examples so the neural network can learn from them and generalize a global solution.
- Unsupervised learning: We use a cost or a score function so the algorithm can learn the patterns that generate the highest score or the minimum cost.
- Semi-supervised learning: By mixing the two other paradigms, we use a small amount of labelled data and a large amount of unlabelled data in order to train a neural network with only a very small set of labeled data.

In this Bachelor work, we will only use the first paradigm.

4.1 About Artifical Neurons (Perceptrons)

Just like in the human brain they are the building blocks of neural networks. A biological neuron is composed with inputs (dendrites) that receive electrical and chemical information. This information is processed by the cell and at the end of the processing a new information is sent trough its outputs (Synaptic terminals)[3].

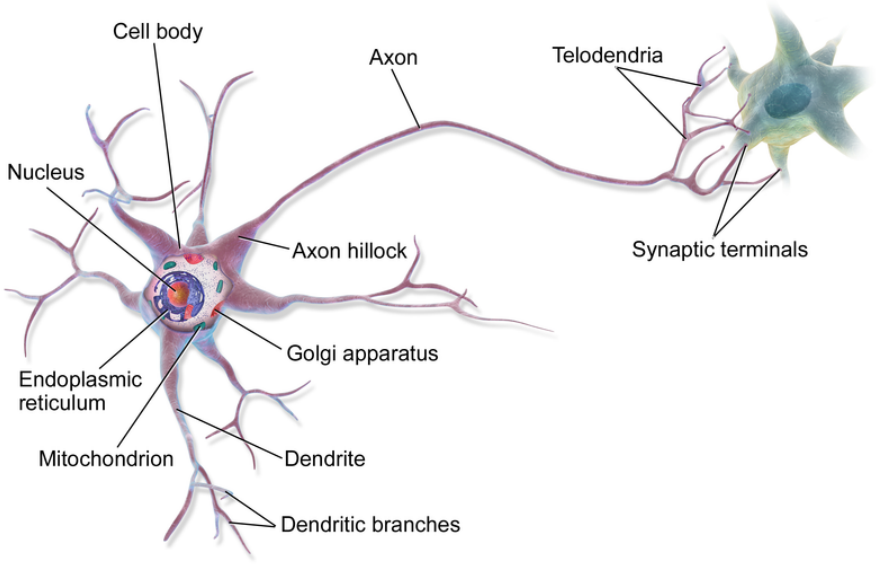

Figure 1: Biological Neuron

An artificial neuron follows the same pattern. Upon a signal is received, it gets multiplied by a weight value for each input. For example, if a neuron has three inputs, each of these three inputs will have its own weight plus one called bias (that will be explained later) that can be adjusted individually. This adjustment is done in the training phase and is based on the error of the last result[4].

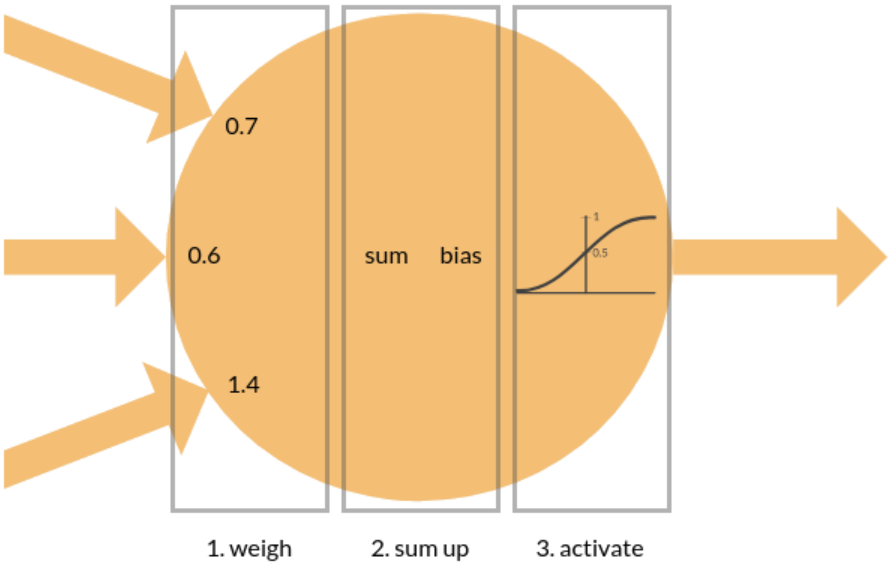

Figure 2: Artifical Neuron

The next step is to sum the weighted signals.
The resulting sum is turned into a signal. This is done by passing the weighted sum to an activation function. The most basic of them is the simple binary function that has only two possible results: The Heaviside step function.

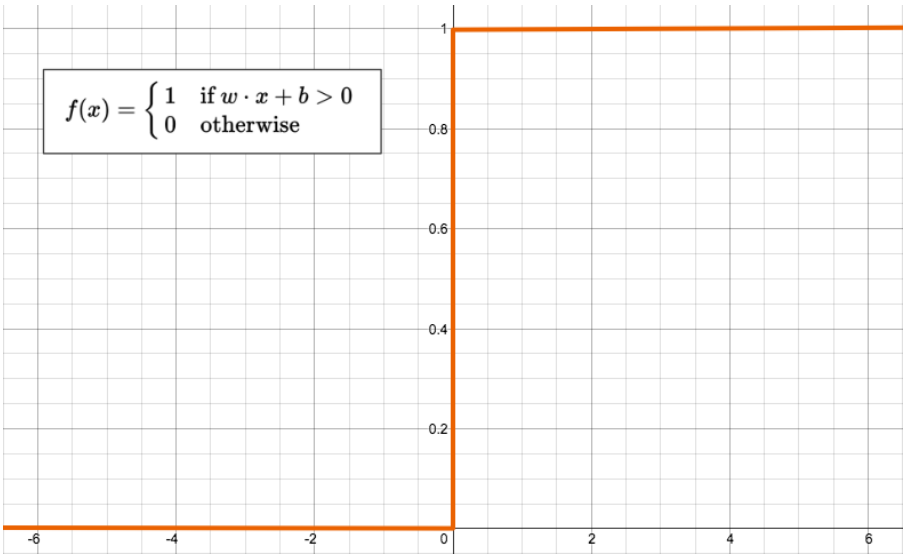

Figure 3: Heaviside Step Function

It returns 1 if the input is positive or zero and zero for negative input. Other activation function like the sigmoid, the hyperbolic tangent or the rectified linear unit (ReLU) can be used as activation as well.

The learning process of an artificial neuron (also called perceptron) works as such:

1. Initialize the weights to small random numbers.
2. Present some inputs to the neuron and compute the output.
3. Proceed to the weights update according to $w_j(t+1) = w_j + \eta(d-y)x_j$ where:
   - d is the desired output;
   - t is the iteration index;
   - x is the input;
   - w is the weights of the perceptron;
   - y is the output of the perceptron;
   - η is learning rate where $0.0 < \eta < 1.0$.

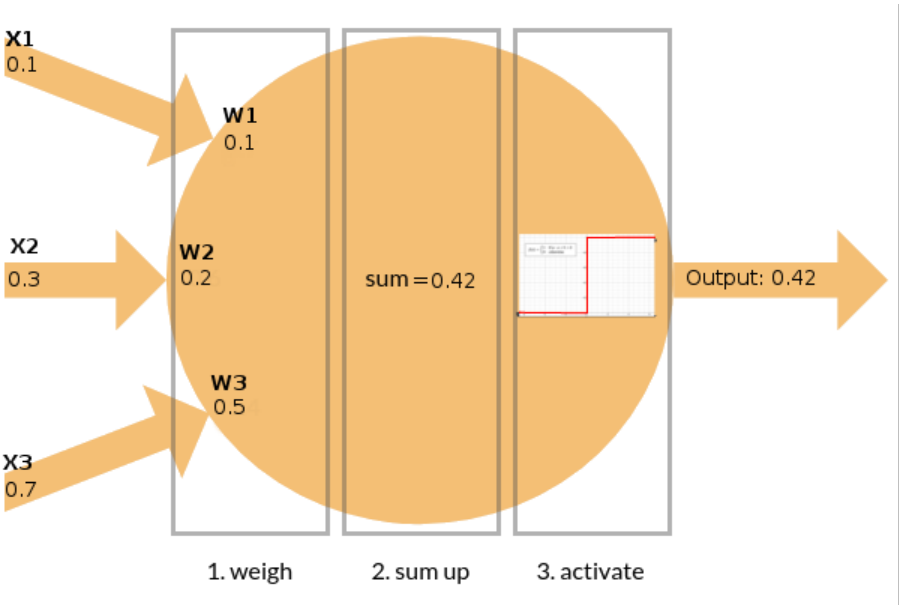

Figure 4: Activation of a perceptron

For example, in the picture above the output of the given values is 0.42. Let's say we wanted 0.6 as a correct output and η is 0.1 and j is 1. The learning process applies like this: $W_3(2) = 0.5 + 0.1(0.6 - 0.42)0.7 = 0.5126$ and so forth for every weights.

Alone a single perceptron can't do much. Actually, it is suitable for only one kind of problem: it can learn only how to classify points on the 2d plane into two different classes (the points of the two classes must be linearly separable). This is because the perceptron's output is in reality a single line in the 2d space, a linear function $ax + b$ that learns the values of a anb b so it can then draw a line to separate the points.

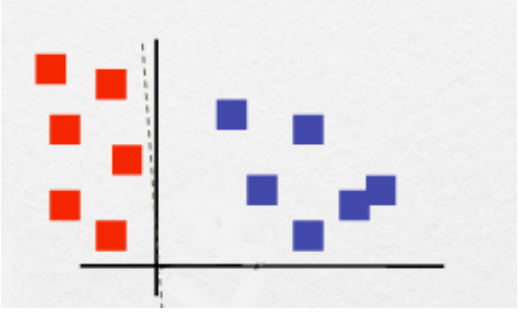

Figure 5: Illustration of a Linear separable problem solved by a perceptron

The weight called bias that is common to all perceptrons gives to their output the ability to shift the output slightly to the left or to the right. That allows the perceptron to have more flexibility and so it can find better fits[5].

We can add multiple perceptrons side by side but we'll just end up drawing more lines so to solve this problem we need multi-layers perceptron.

4.2 About Multi-Layers Perceptron (MLP)

The multi-layers perceptrons solve the problem of non-linear separable problems. An MLP is composed of inputs (input layer), hidden perceptrons (hidden layer: at least one) and output perceptrons (output layer)[6].

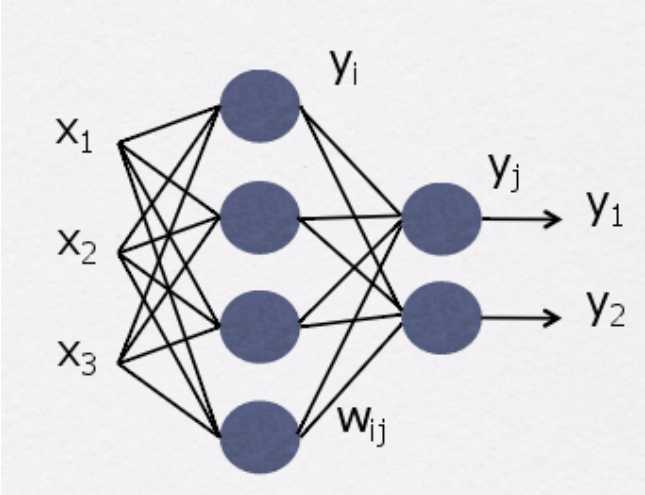

Figure 6: Illustration of a Multi-Layer Perceptron

The MLP learns its weights by a technique called gradient descend[7] and by a process called back-propagation[8].

Gradient descent is an optimization algorithm used to find the parameters values of a function (f) that minimizes a cost (c) function. It is used when parameters of f cannot be found by traditional methods (analytical).

The gradient descent procedure is as follows:

1. The parameters are initialized at 0.0 or a small random value.

$$params = 0.0$$

2. The parameters are then evaluated by using them with f and calculating the cost.

$$c = f(params)$$

3. The derivative of the newly created cost function (c) is calculated. This gives the slope of the cost function at a given point and with the slope the direction (sign) where to move the parameters for the next iteration is known. By doing so we get each time a lower cost function. We'll call this direction *delta*.

$$delta = derivative(c)$$

4. The parameters are then updated by the previously calculated *delta* parameter and a learning rate lr parameter used to control how much the coefficients can change on each update.

$$params = params - (lr * delta)$$

   The points 2 to 4 are to be repeated until the cost is sufficiently low (ideally 0.0).

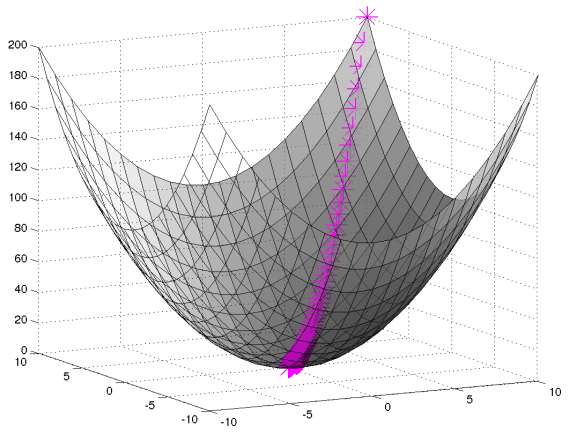

Figure 7: Illustration of gradient descent

Backpropagation is a common method for training neural network.
This procedure is as follows:

1. Compute the error for each perceptron's output using the squared error function and sum them to get the total error.

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

2. We now get to update the weights of the output layer by using the following:

$$w_{oi}^{+} = w_{oi} - \eta(\delta_{oi} output_{hi})$$

where:

- $_i$ is the index of the current perceptron in the current layer;
- $_o$ symbolize the output layer's perceptrons;
- $_h$ symbolize the last hidden layer's perceptrons;
- w is the weight of a perceptron;
- η is the learning rate (between 0 and 1);
- δ is $-(target_{oi} - output_{oi}) output_{oi}(1 - output_{oi})$

3. We now get to update the weights of the hiddens layer by using the following:

$$w_{hi}^{+} = w_{hi} - \eta(\delta_{hi} Input_i)$$

where:

- $_i$ is the index of the current perceptron in the current layer;
- $_o$ symbolize the output layer's perceptrons;
- $_h$ symbolize the current hidden layer's perceptrons;
- w is the weight of a perceptron;
- η is the learning rate (between 0 and 1);
- δ is $-(target_{oi} - output_{hi}) output_{hi}(1 - output_{hi})$
- $Input$ can either be the previous hidden layer of if there is not, the input layer.

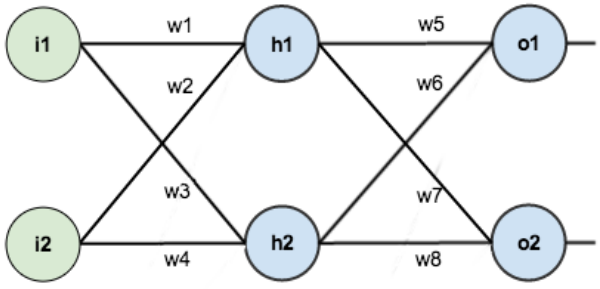

Figure 8: Illustration of a muli-layer perceptron with the same names as in the backpropagation example

4.3 About Convolutional Neural Network (CNN)

One of the most difficult things about training a neural network is giving it the correct features to learn. Feature extraction is sometimes a full time job in the world of machine learning and usually more time is spent on preprocessing the data set than the design or training of the algorithm. With CNNs, the feature extraction is also learnt in the training process by using convolutional layers. Theses filters are feature extractor and they are part of the training of the neural network as they gradually learn to extract useful features. Here are the steps that a CNN usually do in order to do a successful classification[9][10]:

4.3.1 Filtering

A convolution layer is composed of small images called "filters" or kernels. They are usually 3x3 pixels images. At the begining of the training, these images pixel values are randomly set. Then, trough backpropagation, they get ajusted until they find interesting features. Theses filters act like normal filters like the ones used in photoshop like the one for image sharpen for example:

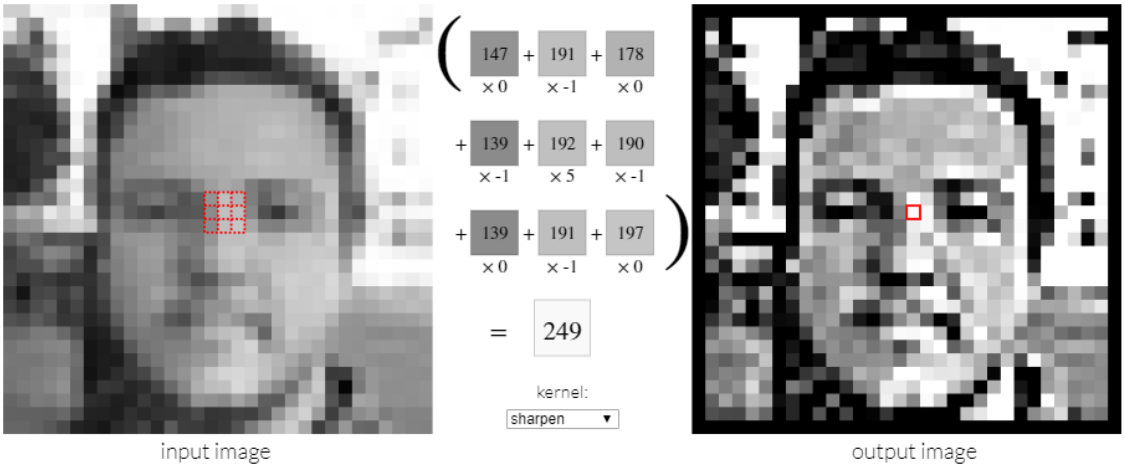

Figure 9: Illustration of filtering

For every filter that filters an input a new filtered image is created. By repeating this process, we gradually extract interesting features: the first layers of convolution only outputs lines and curves, but as we go deeper we begin to see shapes that are common to the training data set like faces, eyes, etc.

4.3.2 MaxPooling

This operation is usually done after every 2 or 3 convolutional layers. The idea is to reduce the input size and extract the most relevant features:

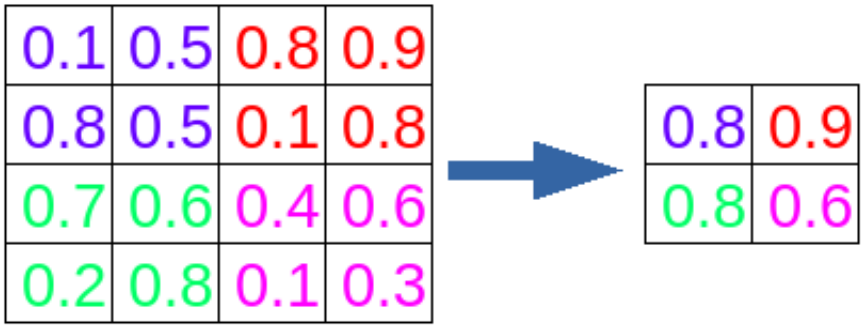

Figure 10: Illustration max pooling

As shown above, the objective is to down-sample an image. It provides basic translation invariance by providing an abstract representation and it reduces the computational cost.

4.3.3 Dropout

The dropout makes the previous part of the convolution network forget a percentage of its weight.

This is done to avoid overfitting (phenomenon when the neural network adjusts too much its weights and comes to a solution that fit perfectly the training set but is completly biased).

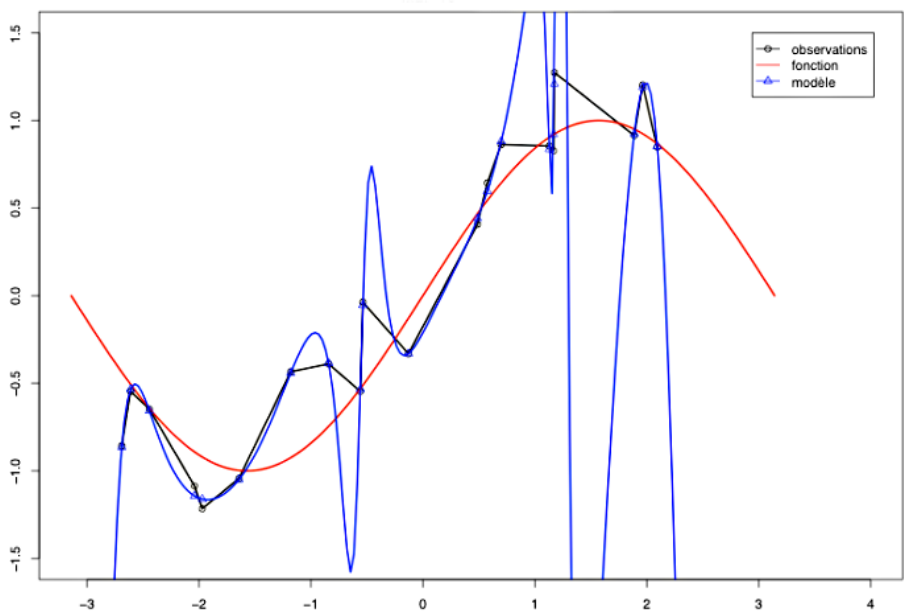

Figure 11: Illustration of overfitting in a MLP with one input, 10 hidden neurons and one output.

4.3.4 Flatten and fully connected

Once all the convolutions are performed, we take the last convolutionnal layer and flatten its filters into a large array that will feed a neural network that is called "Fully Connected Layers".

5 Setup the environment

A setup script is provided for an easy installation as described in the user's manual but we present hereunder the components of the environment as reference.

5.1 Tools and libraries

- Python *3.5*
- Tensorflow-Gpu *1.2.1*
- Keras *2.0.2*
- OpenCV *3.1.0*
- Python Image Library (PIL) *3.4.2*
- Cuda toolkit *8.0.61_375.26*
- cuDNN *5.1.10*
- Matplotlib *2.0.2*
- Numpy *1.12.1*
- Scipy *0.19.0*
- Skimage *0.13.0*
- Sklearn *0.18.1*
- Psutil *5.2.2*
- Anaconda for Python 3.5 *4.4.0*

5.2 Operating system and hardware

In order to make this environment work the following components are needed:

- Ubuntu 14.04.5 LTS
- An Nvidia™ with Compute Capability 3.0 or higher and at least 4G of memory.
- At least 16 GB of ram.
- At least 500 GB of free space if you are planning to do the bias metric experiment.

6 Data and Methods

6.1 About the data set

This Bachelor work will use the 60'000 images and 38 classes of the PlantVillage data set. Each sample is a 256 x 256 RGB colored image. Samples were shot by different people with different cameras and different automatic adjustments under several lightning conditions. Here are a few examples:

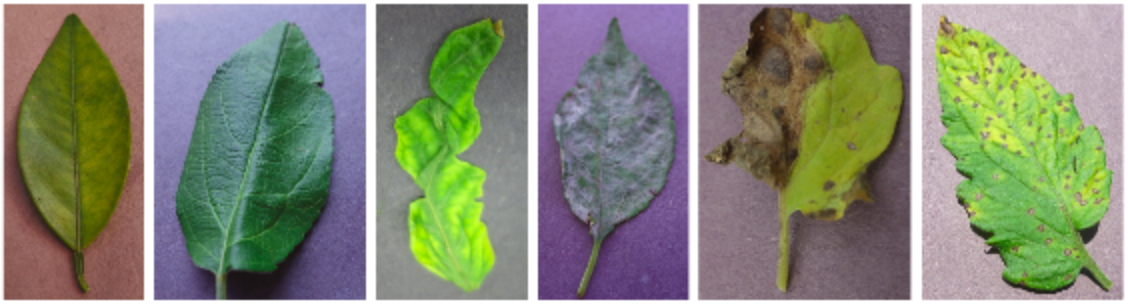

Figure 12: Initial state of the data set

The 38 classes of the data set are distributed as such:

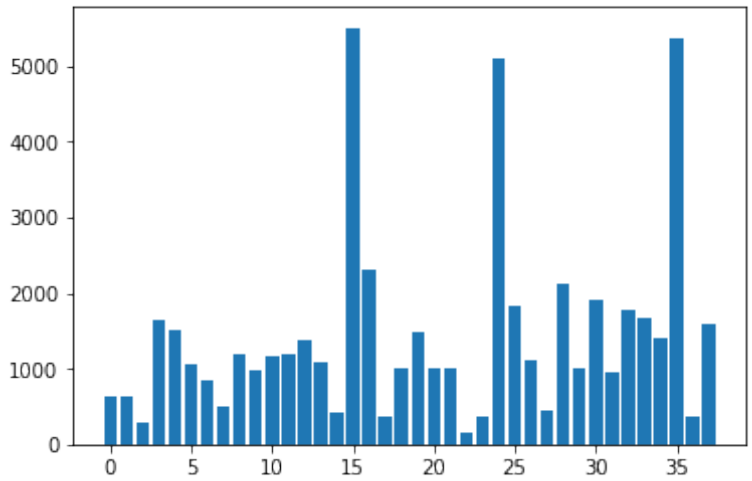

Figure 13: Distribution of the number of samples in the data set per class

The data set has the following classes:

$Name$	$referencenumber$
$Apple\_\_\_Apple\_scab$	0
$Apple\_\_\_Black\_rot$	1
$Apple\_\_\_Cedar\_apple\_rust$	2
$Apple\_\_\_healthy$	3
$Blueberry\_\_\_healthy$	4
$Cherry\_(including\_sour)\_\_\_Powdery\_mildew$	5
$Cherry\_(including\_sour)\_\_\_healthy$	6
$Corn\_(maize)\_\_\_Cercospora\_leaf\_spotGray\_leaf\_spot$	7
$Corn\_(maize)\_\_\_Common\_rust\_$	8
$Corn\_(maize)\_\_\_Northern\_Leaf\_Blight$	9
$Corn\_(maize)\_\_\_healthy$	10
$Grape\_\_\_Black\_rot$	11
$Grape\_\_\_Esca\_(Black\_Measles)$	12
$Grape\_\_\_Leaf\_blight\_(Isariopsis\_Leaf\_Spot)$	13
$Grape\_\_\_healthy$	14
$Orange\_\_\_Haunglongbing\_(Citrus\_greening)$	15
$Peach\_\_\_Bacterial\_spot$	16
$Peach\_\_\_healthy$	17
$Pepper,\_bell\_\_\_Bacterial\_spot$	18
$Pepper,\_bell\_\_\_healthy$	19
$Potato\_\_\_Early\_blight$	20
$Potato\_\_\_Late\_blight$	21
$Potato\_\_\_healthy$	22
$Raspberry\_\_\_healthy$	23
$Soybean\_\_\_healthy$	24
$Squash\_\_\_Powdery\_mildew$	25
$Strawberry\_\_\_Leaf\_scorch$	26
$Strawberry\_\_\_healthy$	27
$Tomato\_\_\_Bacterial\_spot$	28
$Tomato\_\_\_Early\_blight$	29
$Tomato\_\_\_Late\_blight$	30
$Tomato\_\_\_Leaf\_Mold$	31
$Tomato\_\_\_Septoria\_leaf\_spot$	32
$Tomato\_\_\_Spider\_mitesTwo-spotted\_spider\_mite$	33
$Tomato\_\_\_Target\_Spot$	34
$Tomato\_\_\_Tomato\_Yellow\_Leaf\_Curl\_Virus$	35
$Tomato\_\_\_Tomato\_mosaic\_virus$	36
$Tomato\_\_\_healthy$	37

Table 1: data set Classes.

6.2 First model and project layout

In order to discover keras and tensorflow along with other tools and libraries, we first rained from scratch a custom model on PlantVillage the data set.

We started from a mnist example[11] and we progressively adapted it to the PlantVillage data set. During that time we encountered some memory issues: the data set has over 54'000 images of 256x256 pixels each of them having their tree color channels (RGB). We could not fit this ammount of data into the computer's memory so we ended up writing our own data set loader utility that loads n images at the time (usually n is 5'000 but its value can be adjusted to fit the computer's memory). We were aware of the *flow_from_directory*[12] method from the Keras API that allowed us to feed the neural network directly from the data set directory without needing to write any code but we implemented our own because we wanted to do some custom preprocessing on our data and the main reason is that we can reuse this utility for other experiments.

The custom model reached an accuracy of 0.87901 % with a loss of 0.40690 over ten epochs with a learning rate of 0.0001. It was then we realised we needed the state of the art for this classification problem.

In the end we developped some other tool that form our final project:

- img_loader: The data set loading tool that we introduced earlier.
- bias_metric: This is the module for computing bias in the data set. This experiment will be described in the sections bellow.
- heatmap_generate: This is the module for computing heatmaps on a large scale. Heatmaps will be described in the next sections.
- img_processing: This module is the image processing core of this project. This is also the place where we generate the two new backgrounds. The backgrounds experiments will be described in the next section.
- model_utils: Functions for training and evaluating our data sets with the img_loader tool.
- VGG16_ft: A tool to instantiate a model with VGG16 architecture. VGG16 will be described in the next sections.

6.3 Background experiments

This data set contained a lot of bias because instead of learning the disease features on the leaves the CNN could instead learn the color differences between each class and use this bias feature in order to do its classification.

Thanks to the work of mister Boris Conforty[13] this data set has already been processed to remove such bias.

Figure 14: Final state of the data set

But then the question of the background arose. Is it creating bias? This is a legit question to ask as there are no direct evidence showing that the CNN learned the corrects features (which are on the leaves). To answer this question we used a technique called Class Activation Mapping (which will be described in the next section).

Four types of background have been selected for this experiment:

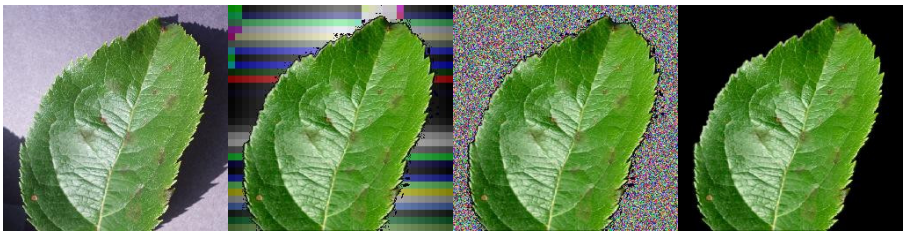

Figure 15: Four kinds of background were used.

First, there is the default laboratory background. Next we generated *noisy* backgrounds by creating images with random pixels on them. The idea here was to see if a CNN could improve its performances by putting noise in the background. Next we have the *artistic* background. Thanks to the work of Andrej Bauer[14], we were able to generate images with some abstract shapes in it. The goal here was to see how the CNN would handle background shapes as the original background does not have any. Finally, we also used the black background that was provided with this data set in order to *force* the CNN to only learn the features on the leaf.

6.4 Colored cats and dogs

We use a data set about cats and dogs[20] just to verify and see by ourselfs how much bias a bad lighting condition would cause.

Figure 16: Cats and dogs colored data set.

We trained our VGG16 fined tuned model (which is described in the next sections) with this data set and it took only two epochs to reach an accuracy of 100 % and a loss of $7.58 * 10^{-6}$. But as soon as we swapped theses colors the CNN predicted dogs as cats and cats as dogs. So with that we can proove that the classes have to have the same lightning conditions in order to remove this bias.

6.5 What is a Class activation Mapping (CAM)?

A class activation mapping is a score map of the CNN kernels outputs for a given class. It means its a map of how much attention the CNN is paying to each pixels of the leaf.

6.6 CAM algorithm

The traditional fully connected layer shall be removed and instead will be added a Global Average Pooling layer (GAP). The network is then to be trained. Once it's done, an image has to be fed to the neural network and the outputs of the last convention layer shall be recovered, resized and weightened by the GAP's weights that correspond to the class we want to draw its activation [15].

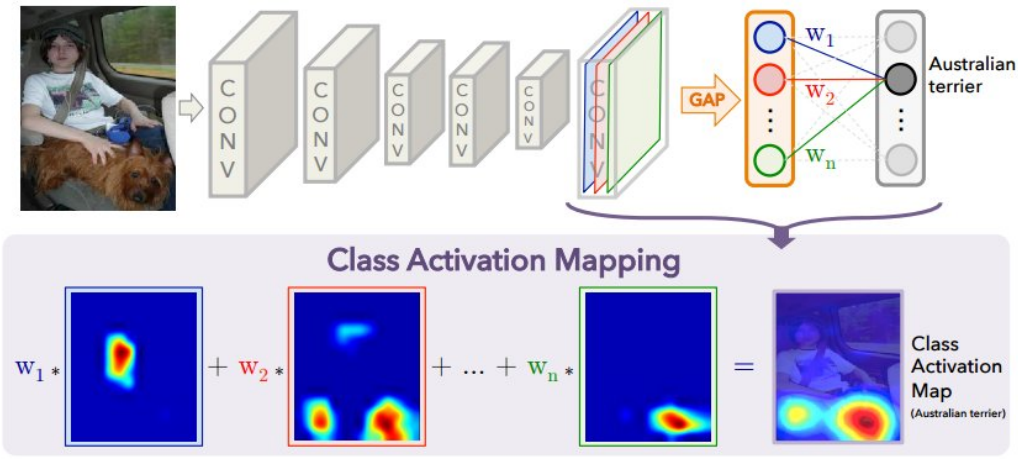

Figure 17: Class Activation Mapping creation process.

By doing so we can see where the CNN is *looking*, what features are the most useful to do classification. The CAM itself is the weighted sum of the activation of the CNN's kernels. In other words, the higher one value of this map is the more important it was to achieve classification.

6.6.1 CAM implementation

In order to compute quickly a large number of class activation mappings (this was a requirement for the bias metric tool), the Tensorflow[16] runtime environement was used to run the operations directly on its distributed and GPU accelerated computation graph.

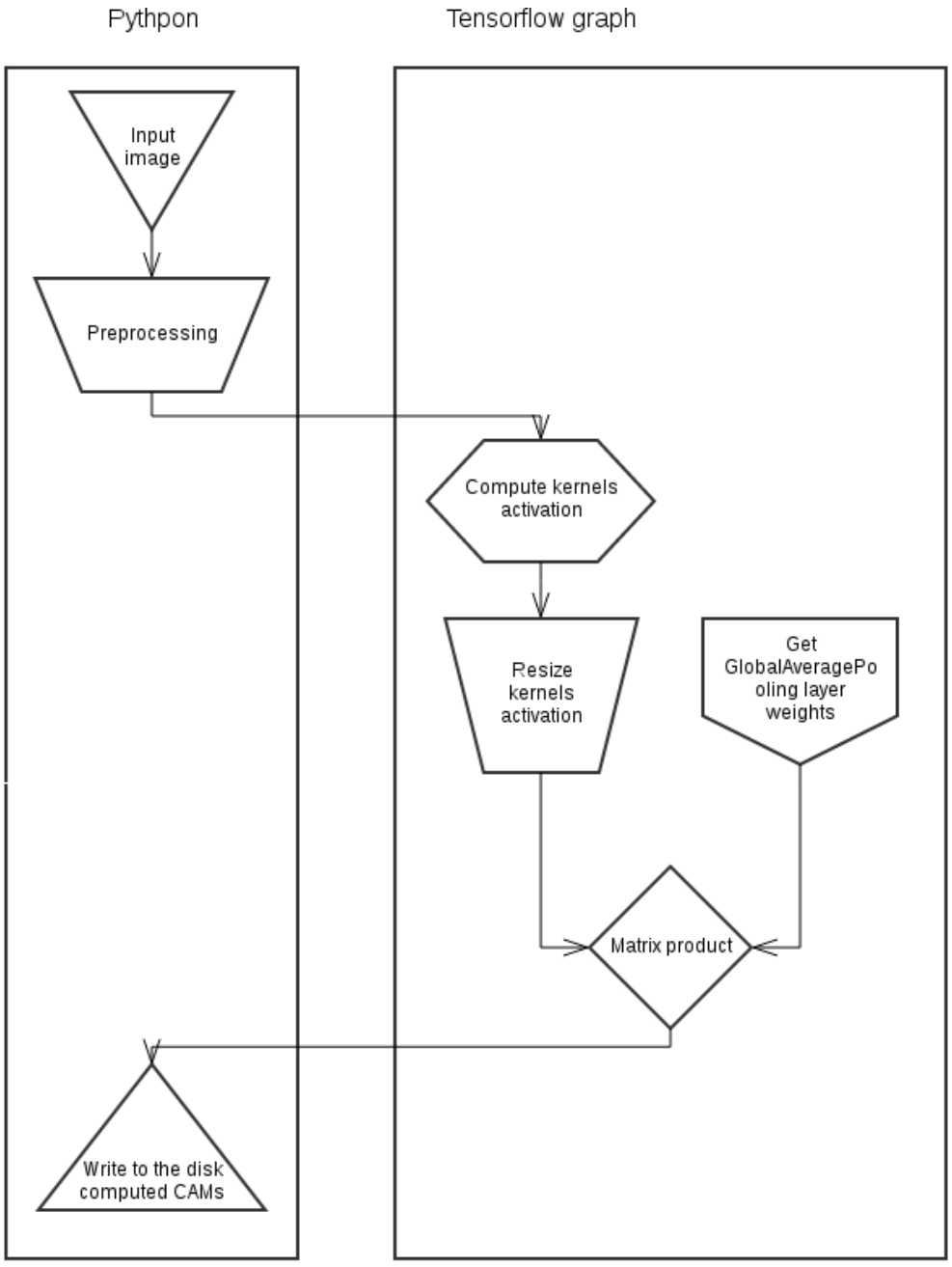

Figure 18: Class Activation Mapping final implementation.

As you can see above, most of the heavy computation is done on Tensorflow. A previous implementation was doing the same with OpenCV2[17] but it was 10 times slower. For the Tensorflow implementation it takes only 0.5 seconds to output a CAM vs 10 seconds for the OpenCV2 implementation.

6.6.2 CAM implementation's validation

In order to validate our implementation, we used the same network we used to classify leaves and we trained it on a data set called Caltech 101 [18]. It's a data set about various objects like cars, watches, planes, etc. With a total of 101 classes it was good enough to test in 'real conditions' our implementation. With 10 epochs our classifier reached 90% accuracy on validation set and we decided it was good enough to draw class activation mapping as for the CNN to achieve these performances it had to learn at least some relevant features.
We plotted some CAMs to ensure that they were correctly generated:

Figure 19: Colored class activation mappings from clatech 101 data set.

As we can see on theses images, the most relevant feature on the plane image is the wing and on the watch this is the dial.

6.7 VGG16

For theses experiments a very good model has been used: VGG16[19]. From the Visual Geometry Group (Department of Engineering Science, University of Oxford), it has won the Imagenet Large Scale Visual Recognition challenge 2014. This model is very strong when it comes to object recognition and is perfectly suited to our needs. Indeed, it has been already trained so we can use its convolutional weights to speed up our model training and by using its weights we also generate a model that is also strong in object recognition. When it comes to drawing a class activation mapping it is handy to have a model that look only at one object in the image so we can output very clear CAMs.

$Layer(type)$	$OutputShape$	$NumberofParam$
$block1\_conv1(Conv2D)$	$(None, None, None, 64)$	1792
$block1\_conv2(Conv2D)$	$(None, None, None, 64)$	36928
$block1\_pool(MaxPooling2D)$	$(None, None, None, 64)$	0
$block2\_conv1(Conv2D)$	$(None, None, None, 128)$	73856
$block2\_conv2(Conv2D)$	$(None, None, None, 128)$	147584
$block2\_pool(MaxPooling2D)$	$(None, None, None, 128)$	0
$block3\_conv1(Conv2D)$	$(None, None, None, 256)$	295168
$block3\_conv2(Conv2D)$	$(None, None, None, 256)$	590080
$block3\_conv3(Conv2D)$	$(None, None, None, 256)$	590080
$block3\_pool(MaxPooling2D)$	$(None, None, None, 256)$	0
$block4\_conv1(Conv2D)$	$(None, None, None, 512)$	1180160
$block4\_conv2(Conv2D)$	$(None, None, None, 512)$	2359808
$block4\_conv3(Conv2D)$	$(None, None, None, 512)$	2359808
$block4\_pool(MaxPooling2D)$	$(None, None, None, 512)$	0
$block5\_conv1(Conv2D)$	$(None, None, None, 512)$	2359808
$block5\_conv2(Conv2D)$	$(None, None, None, 512)$	2359808
$block5\_conv3(Conv2D)$	$(None, None, None, 512)$	2359808
$block5\_pool(MaxPooling2D)$	$(None, None, None, 512)$	0
$CAM(Conv2D)$	$(None, None, None, 512)$	2359808
$GAP(GlobalAveragePooling2D)$	$(None, 512)$	0
$W(Dense)$	$(None, 38)$	19494

Table 2: Architecture of VGG16 modified to generate Class Activation Mapping

6.8 Bias metric tool experiments

To quantify bias in a data set, we have to proceed as such: First we have to compute class activation mappings for every samples in the test data set and store them as floating point values (like in TIFF format). Then we do the sum of every class activation mapping's values in two areas: inside and outside the leaf. We then compute the metric with this formula: $score = outside/(outside + inside)$

Figure 20: Bias process, outside and inside shapes for one sample

This gives us how much attention the CNN is paying to the outside of the leaf ratio (the looking error ratio) for every sample of the test data set.

7 Experiments and results

7.1 Background experiments results

Here are the results of the four backgrounds experiment. We fine tuned VGG16 with each data set with the following parameters:

- Number of epochs: 5
- Learning rate: 0.0001
- Momentum: 0.9

The data set was first randomly shuffled, then 75 % of it was dedicated to training and the remaining 25% was used for validation. Here is the performance of each model on the validation data set:

Background	*loss*	*accuracy*
normal	0.13166247157077818	0.96366684765972987
random	0.096651912232993131	0.97121296294112658
art	0.099246639476883308	0.96869759119176679
black	0.03794590758516931	0.98798546810395294

Each trained version of VGG16 (or model) was then fed with one sample of every class for every couple of models and input (trained with normal background fed with normal background, trained with normal background fed with random background, etc...).

7.1.1 CAMs with normal training and normal inputs

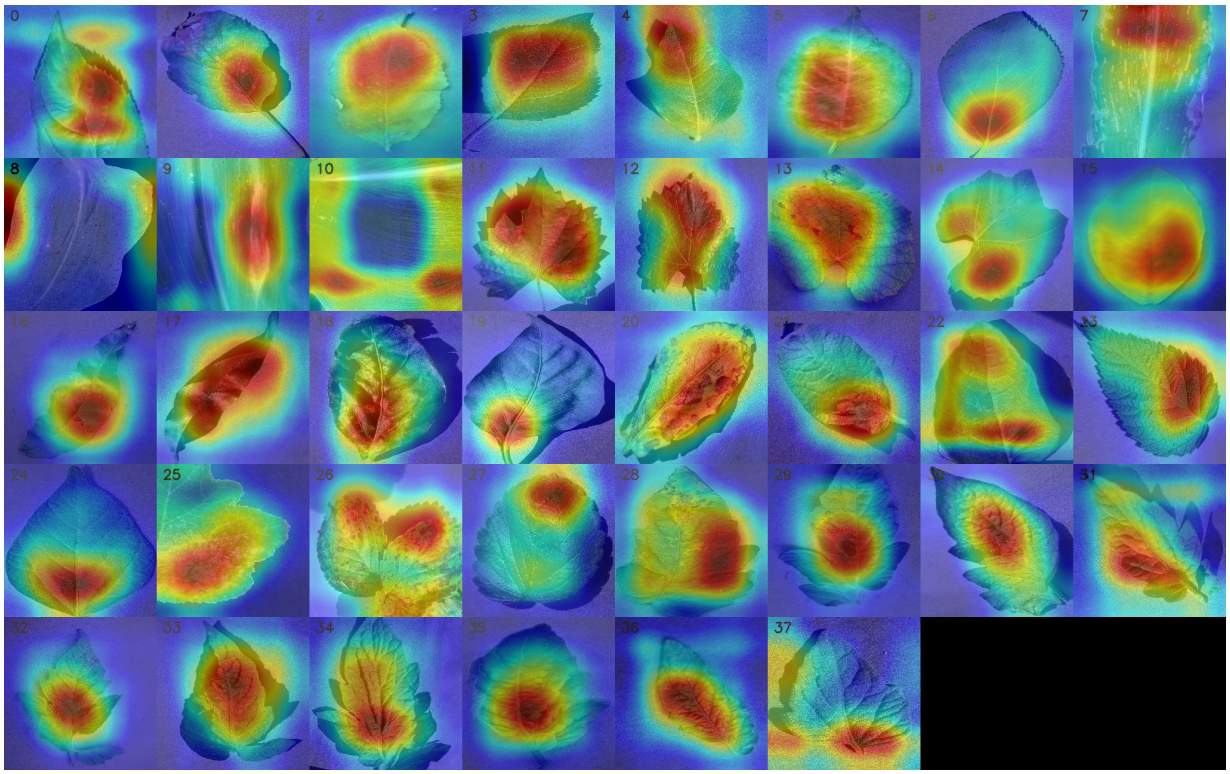

Figure 21: The class activation mappings normal to normal.

7.1.2 CAMs with normal training and random inputs

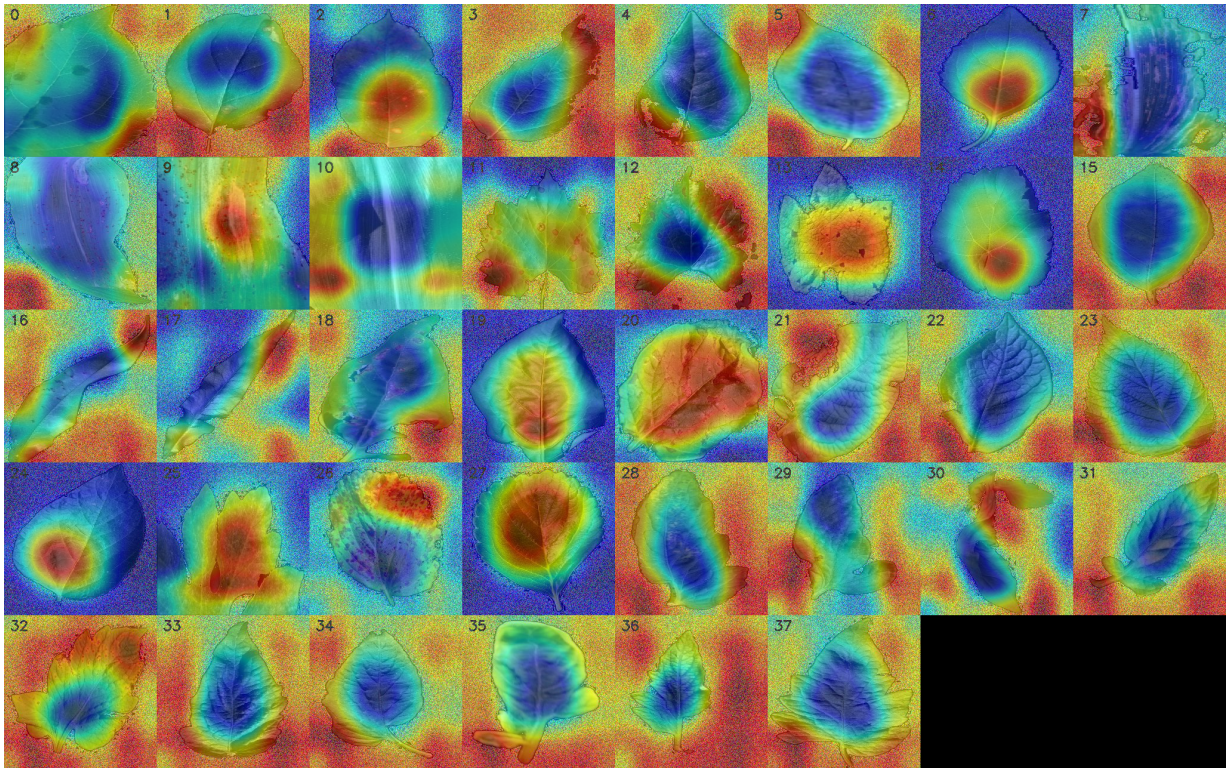

Figure 22: The class activation mappings normal to rand.

7.1.3 CAMs with normal training and art inputs

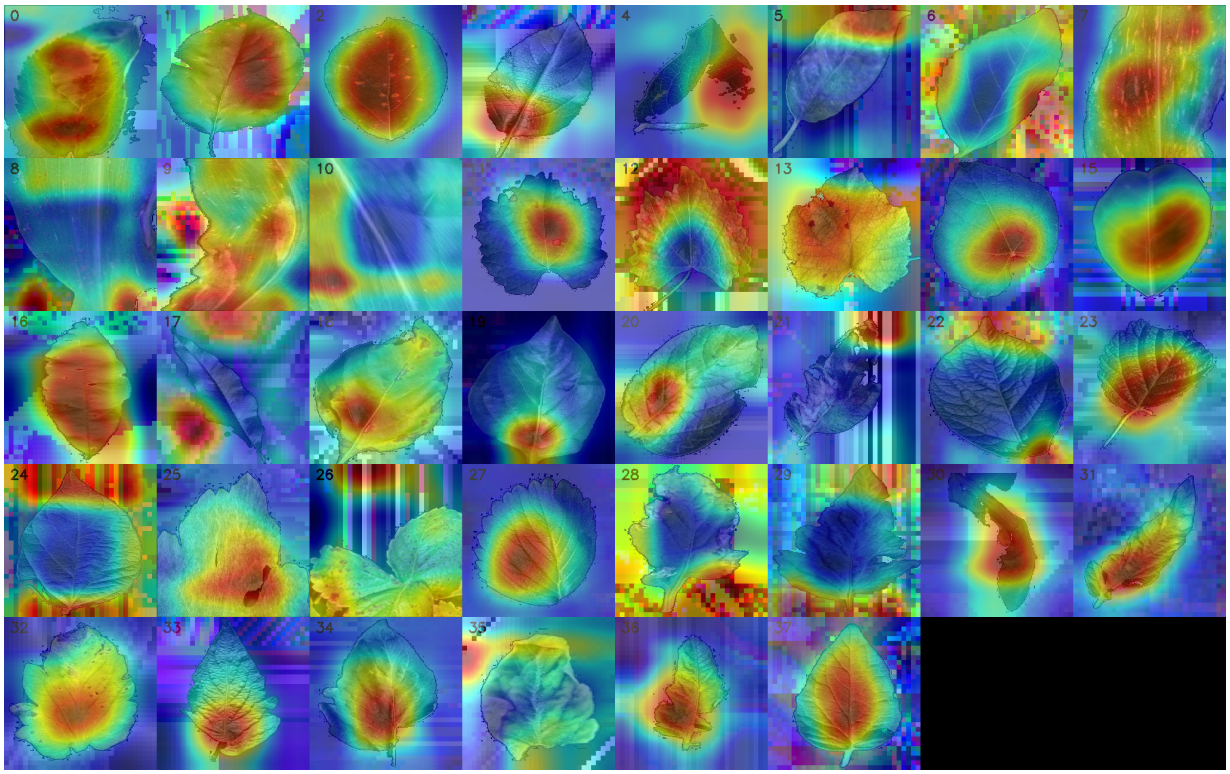

Figure 23: The class activation mappings normal to art.

7.1.4 CAMs with normal training and black inputs

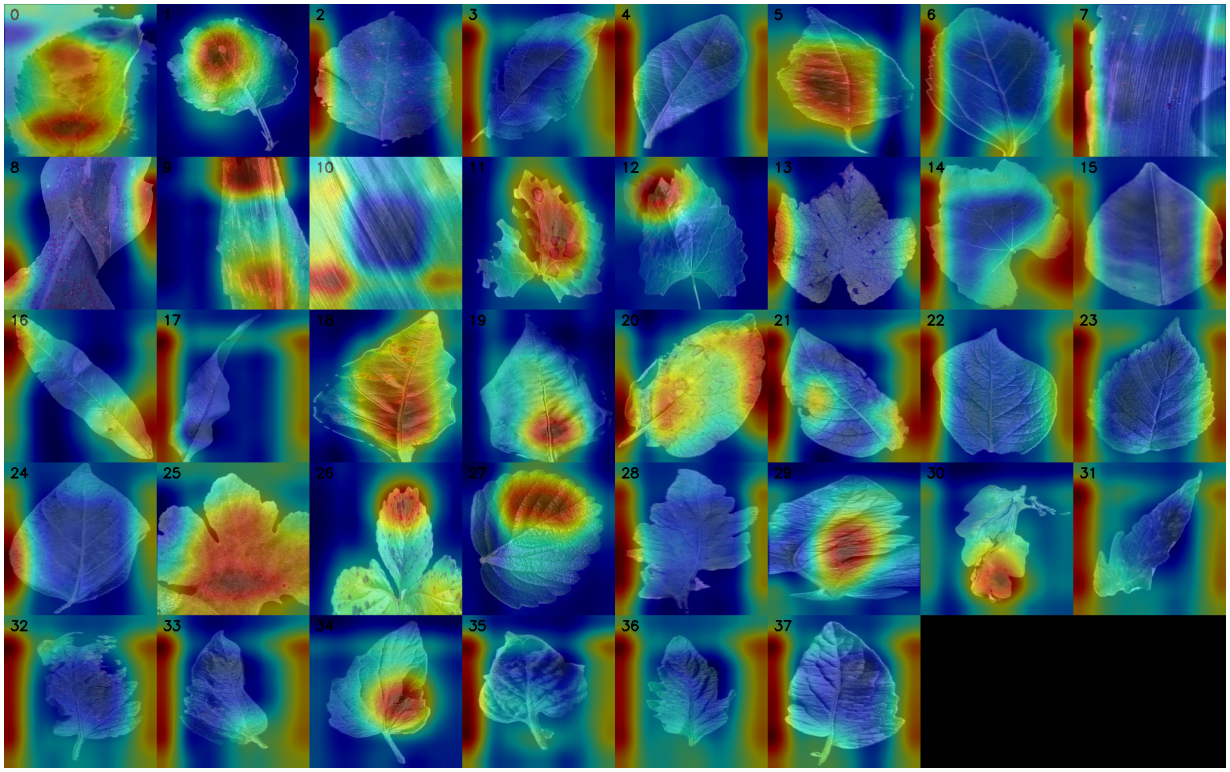

Figure 24: The class activation mappings normal to black.

7.1.5 CAMs with random training and normal inputs

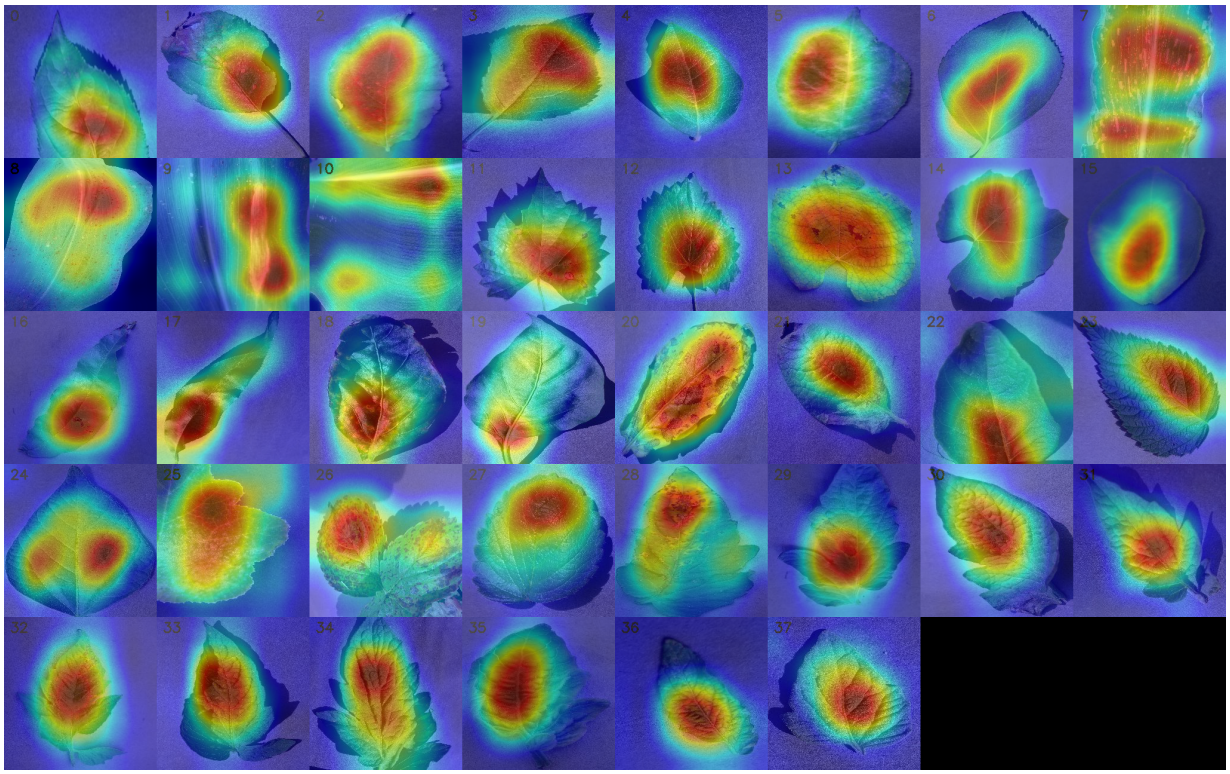

Figure 25: The class activation mappings rand to normal.

7.1.6 CAMs with random training and random inputs

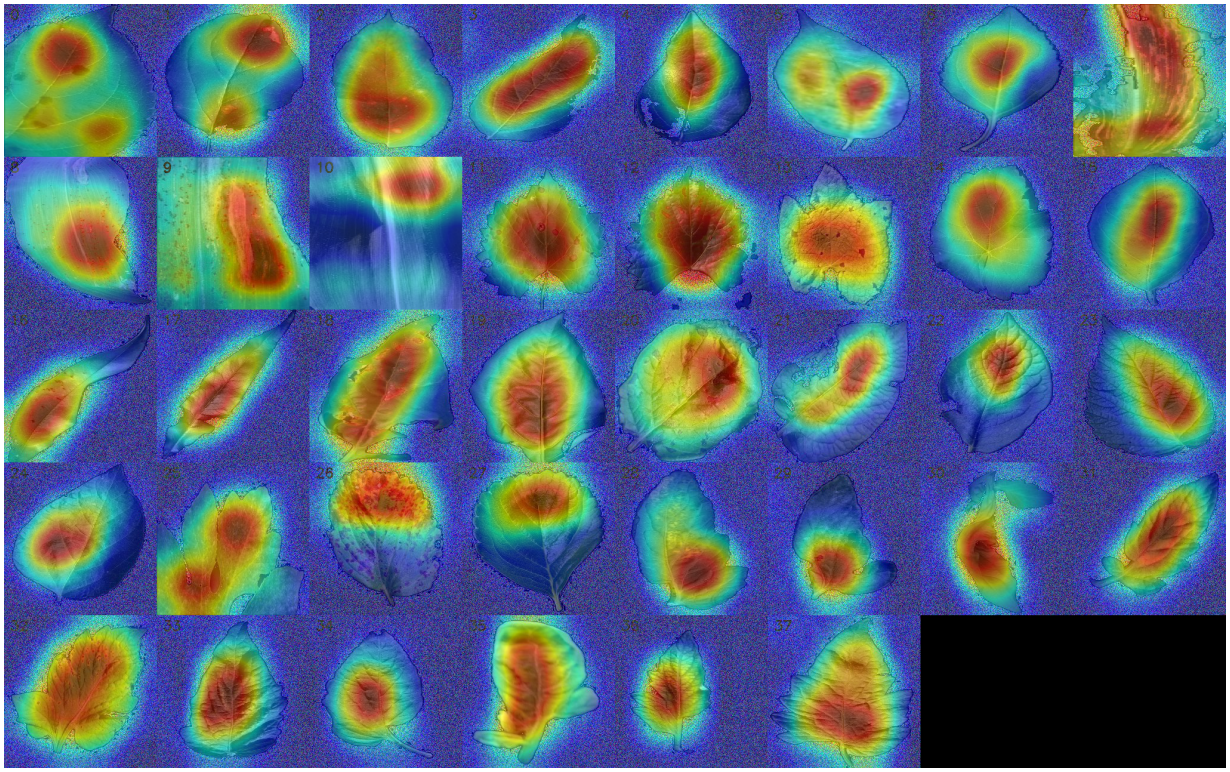

Figure 26: The class activation mappings rand to rand.

7.1.7 CAMs with random training and art inputs

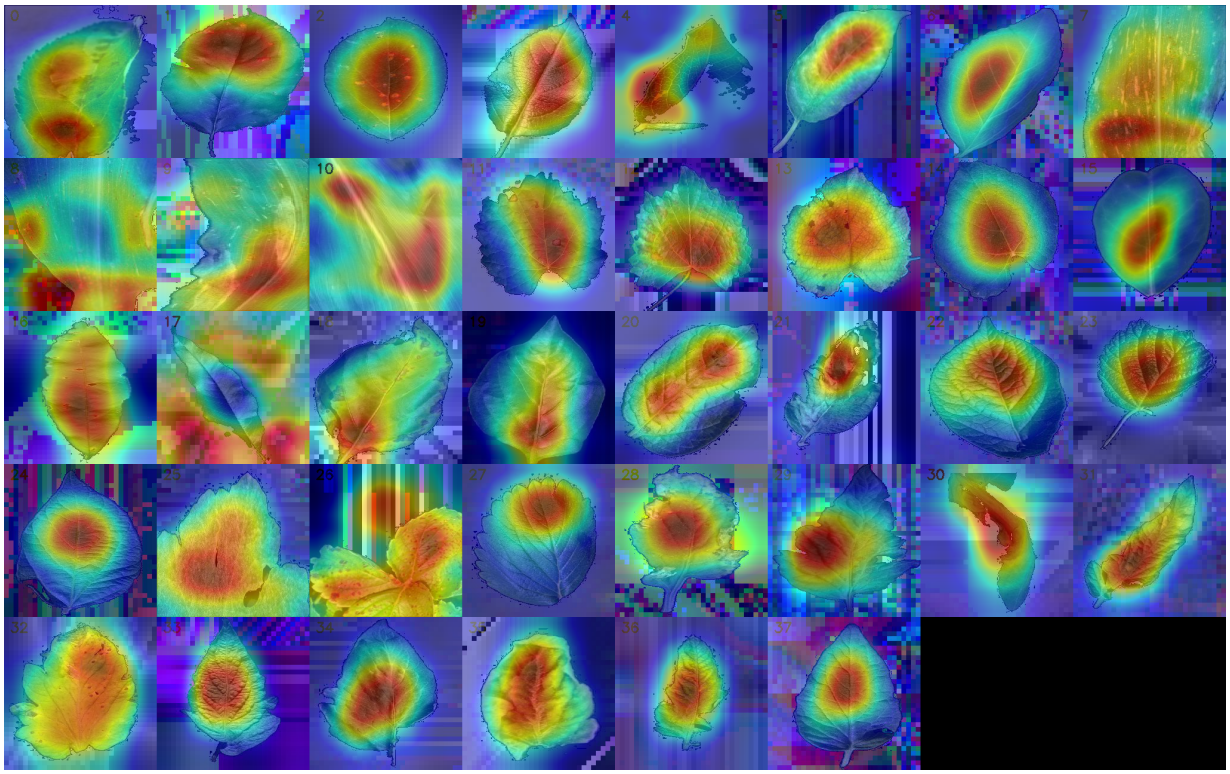

Figure 27: The class activation mappings rand to art.

7.1.8 CAMs with random training and black inputs

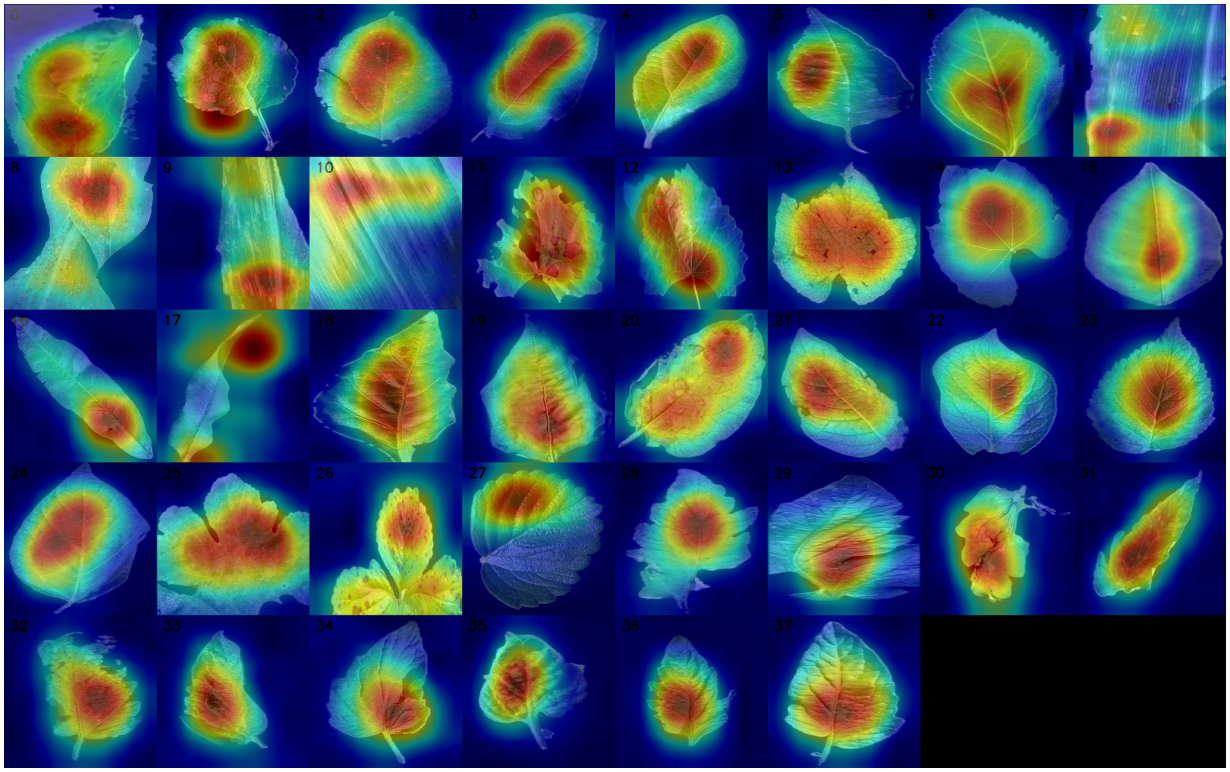

Figure 28: The class activation mappings rand to black.

7.1.9 CAMs with art training and normal inputs

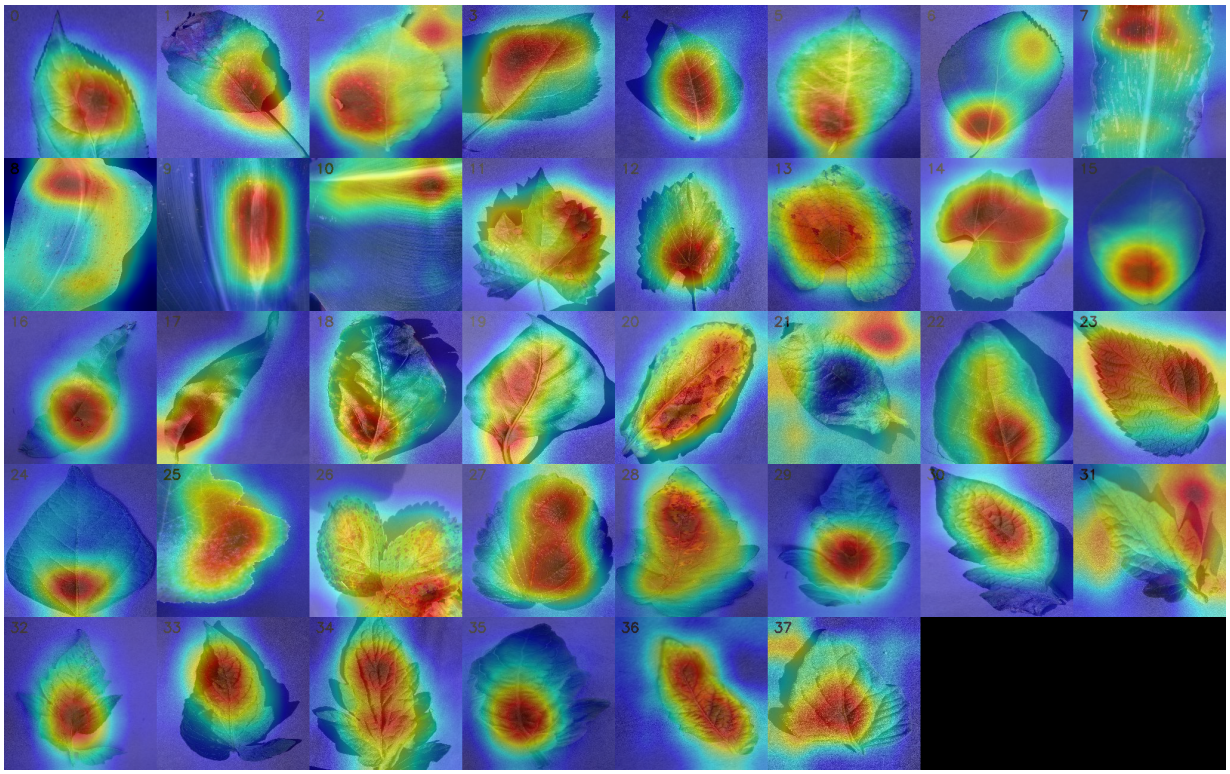

Figure 29: The class activation mappings art to normal.

7.1.10 CAMs with art training and random inputs

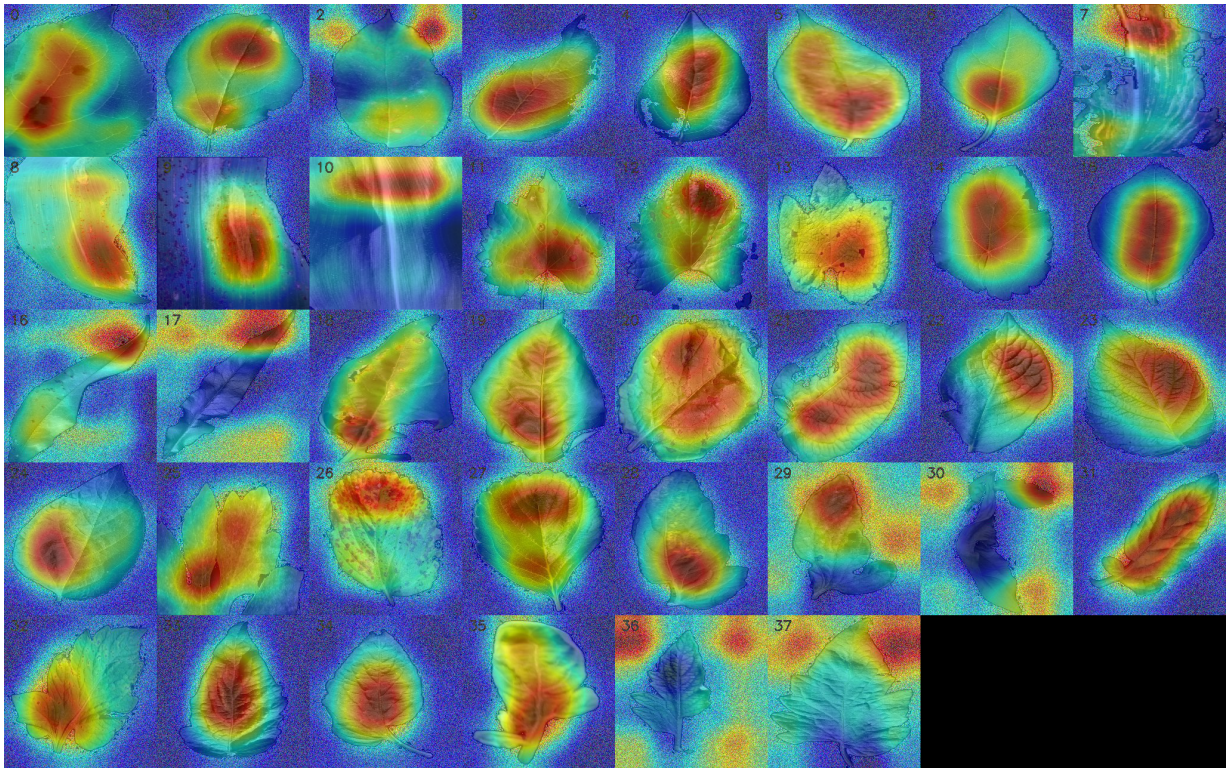

Figure 30: The class activation mappings art to rand.

7.1.11 CAMs with art training and art inputs

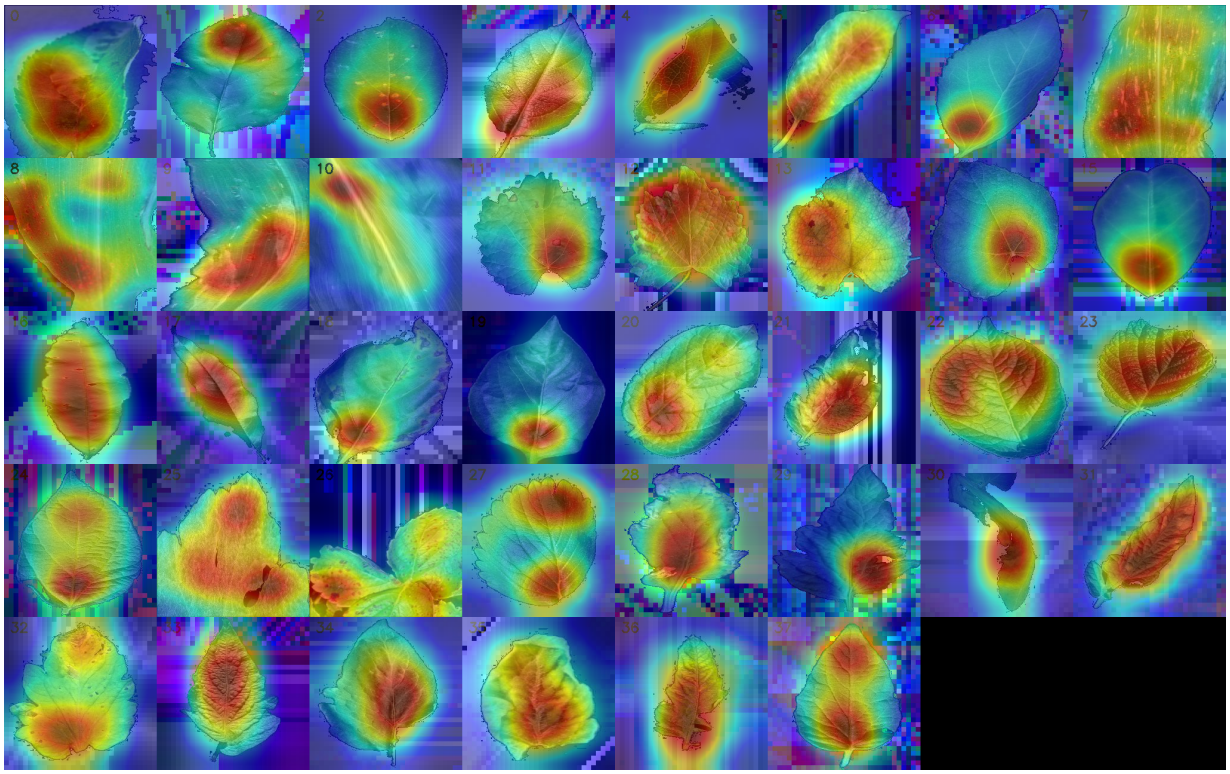

Figure 31: The class activation mappings art to art.

7.1.12 CAMs with art training and black inputs

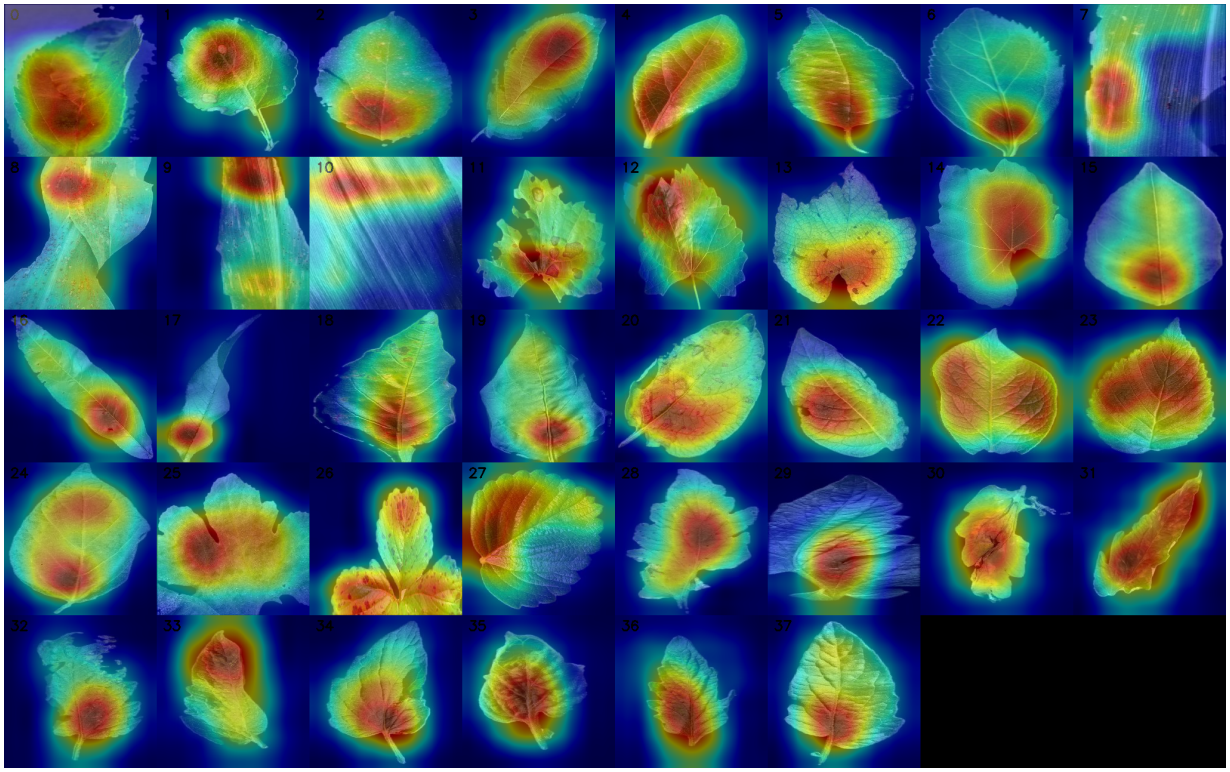

Figure 32: The class activation mappings art to black.

7.1.13 CAMs with black training and normal inputs

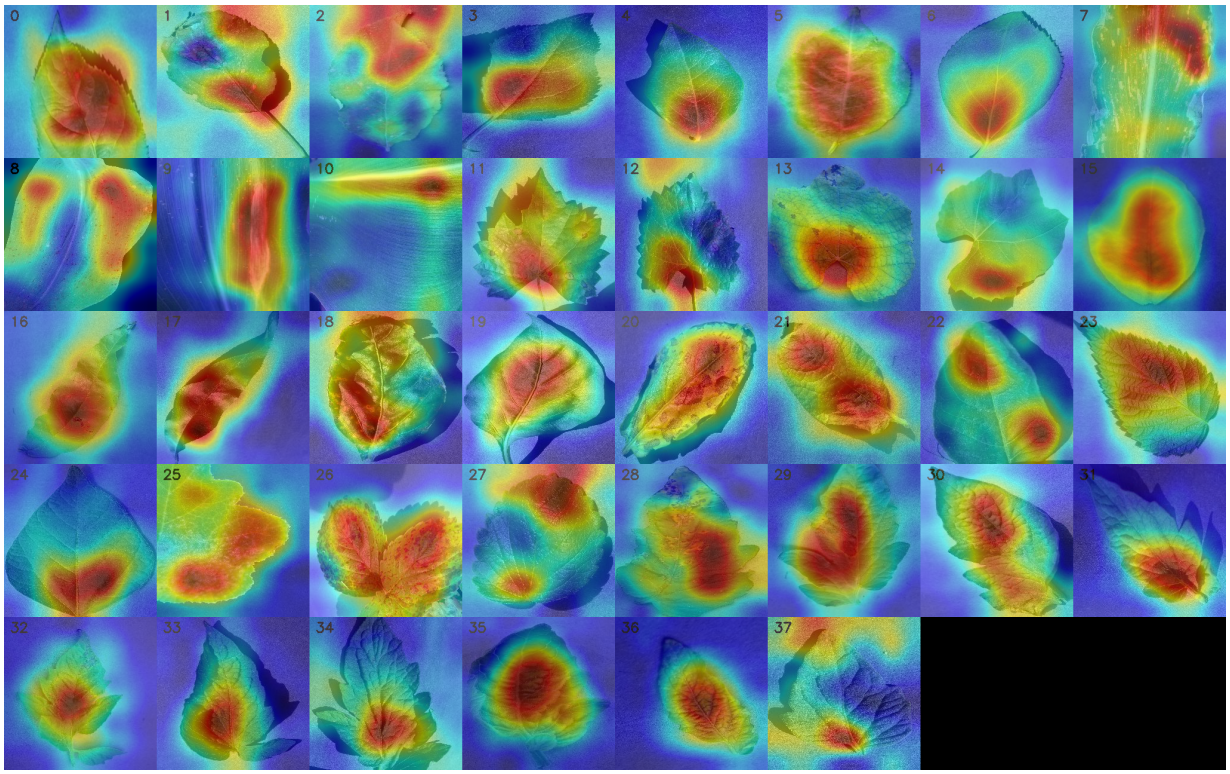

Figure 33: The class activation mappings black to normal.

7.1.14 CAMs with black training and random inputs

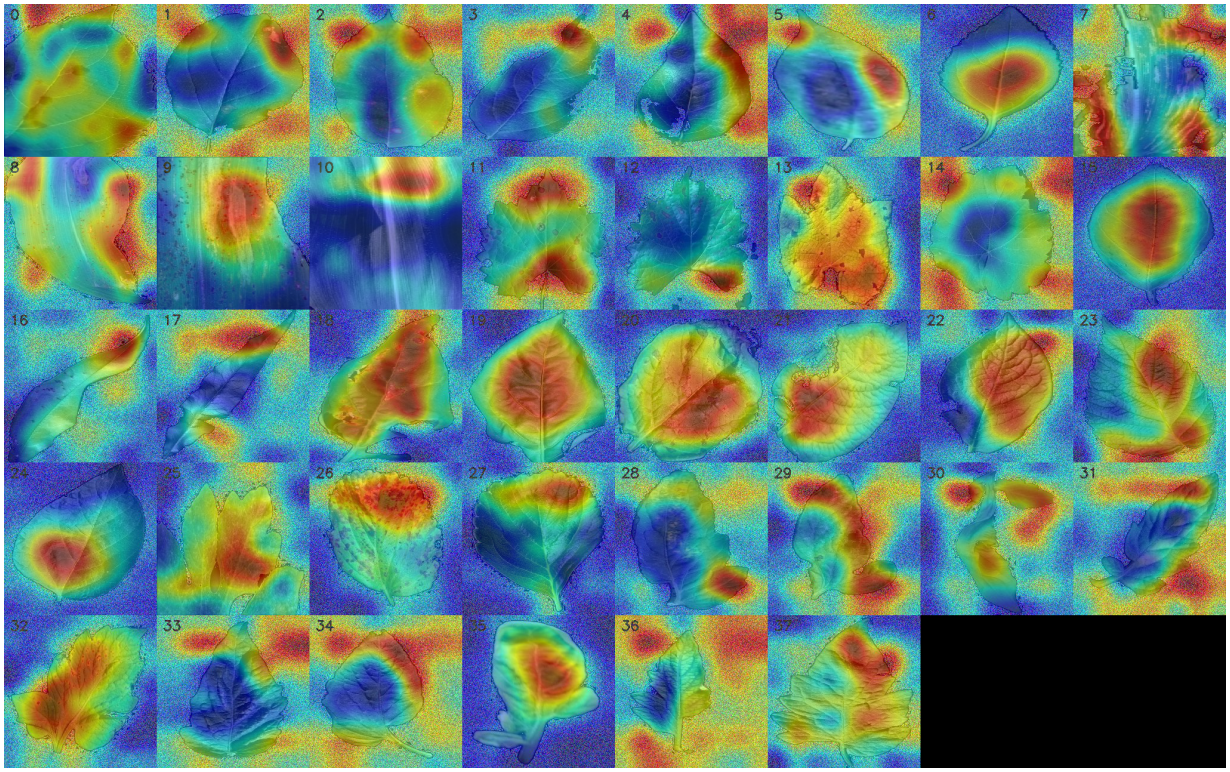

Figure 34: The class activation mappings black to rand.

7.1.15 CAMs with black training and art inputs

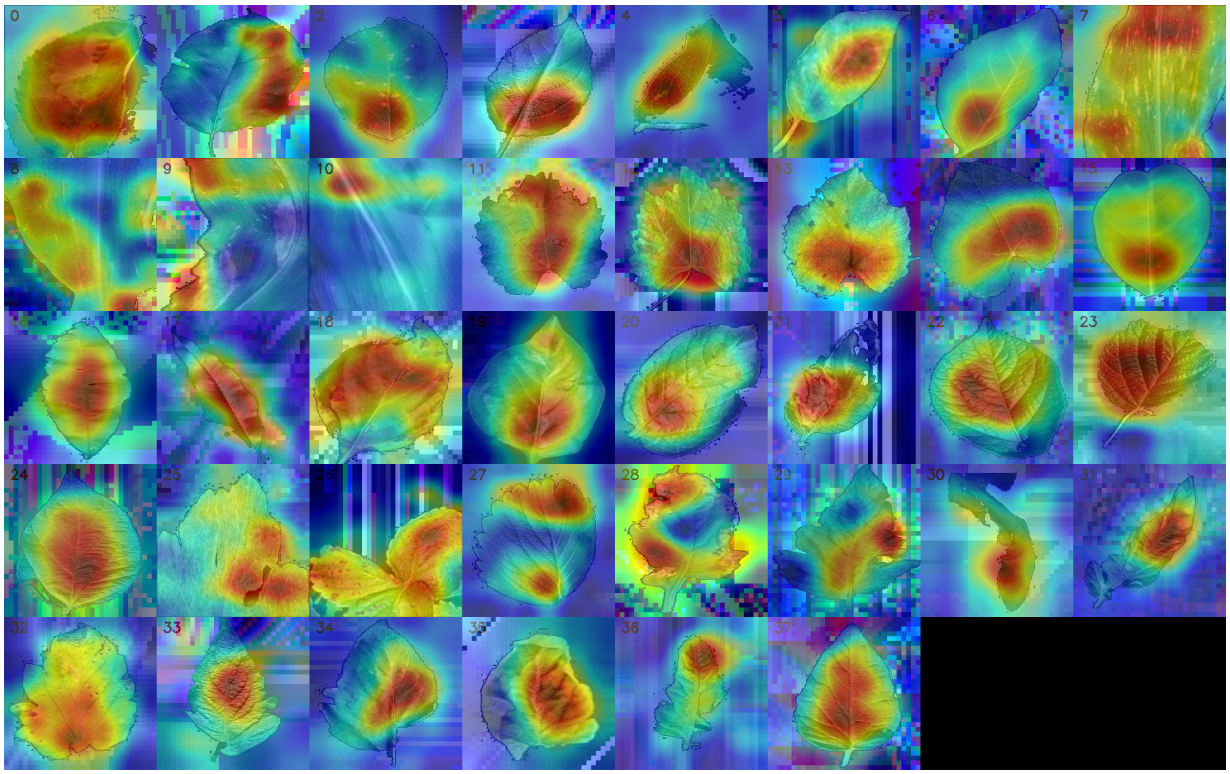

Figure 35: The class activation mappings black to art.

7.1.16 CAMs with black training and black inputs

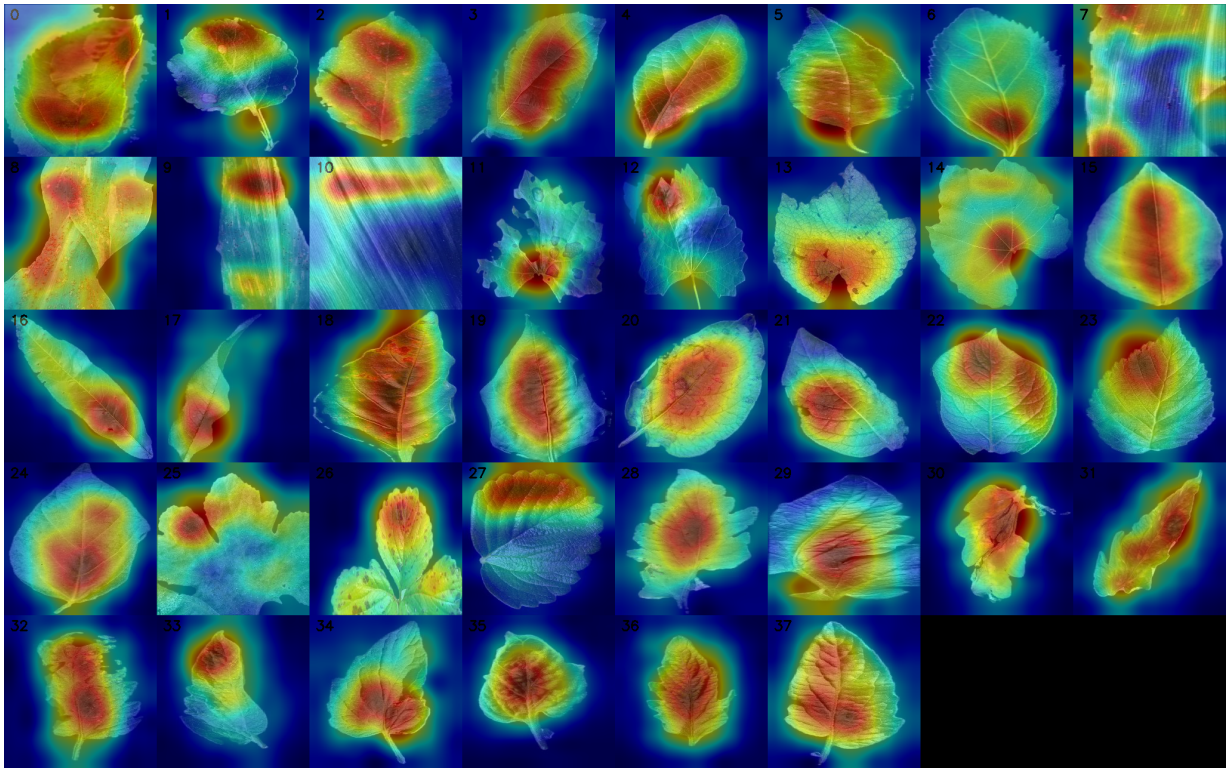

Figure 36: The class activation mappings black to black.

7.2 Bias experiments results

For this experiment, we didn't know if we had to normalize the data in order to get rid of negatives values that where on our class activation mappings. So we computed the metric as presented in the section 4 for untouched values and nomralized (between 0 and 1) values.

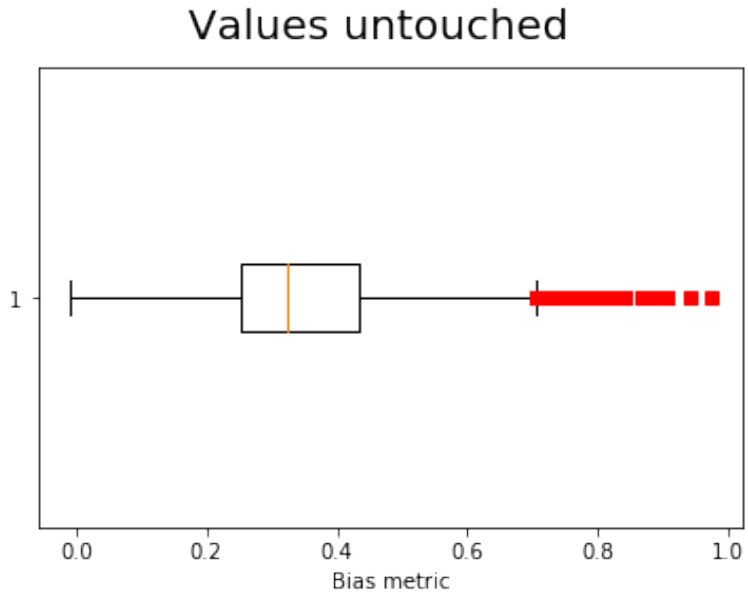

Figure 37: Distribution of the bias metrics results.

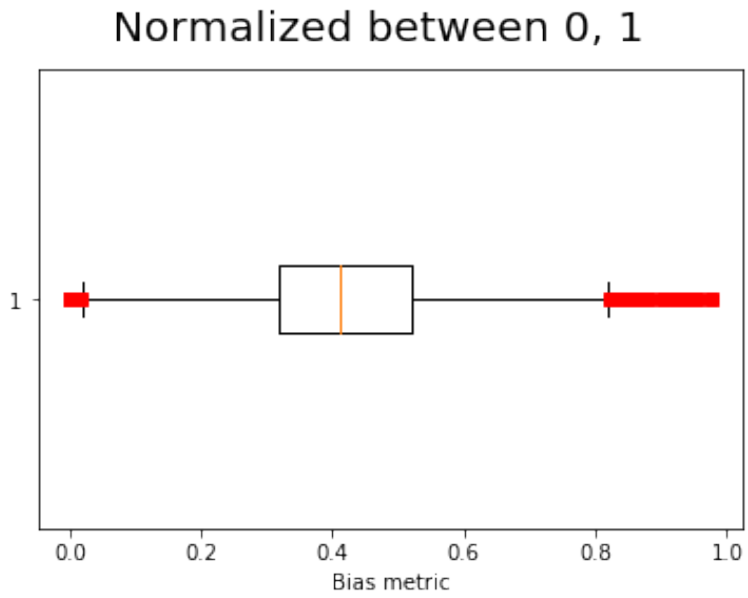

Figure 38: Distribution of the bias metrics results normalized between 0 and 1.

8 Analysis

8.1 Background experiments

The results will be analysed by training method.

1. *Normal training*: As we can see on the results, the CNN has a hard time with the randomly generated background and the art background. My assumption here is that the CNN has learnt somes features on the background of the data set and they induce it in error as its kernels match with parts of the art background or the random background. The black baground input is also very interesting. It makes the CNN focus on the border of the image maybe because there were some part the the data set that had also theses kind of black border and maybe this is why it draws its attention here: Because one vertical line kernel was important and it is found in these spots.

2. *Random training*: This kind of training seems to be the state of the art. No matter what background we use, the CNN's attention stays on the leaf. My guess here is that the CNN's kernels have learnt to look only at the leaf. Because of the randomness near the outline of the leaf and in the background, the only option that was left for the CNN was to learn patterns on the leafs themselfs. That is why it is not affected by backgrounds anymore.

3. *Art training*: Like the random training, the art training gave the same kind of results but still fails when it has to deal with random background images. My guess here is that this training was serving the same purpose than the Random training but it gave less performances due to the fact that the art backgrounds follow a more predictable pattern than the random ones.

4. *Black training*: With normal background inputs, the CNN reacts to the leaf shadow as we can see on some classes. So by adding black background to our data set we induced bias as the CNN's kernels trigger on the leaf's shadow. It also fails the random part but surprisingly it handles quite well the art part. Maybe it's because there is not much 'pure' black in the art backgrounds. So by knowing only one color as backgrond it reinforced itself against the case of when the training background color is not in the input blackground colors.

So we can conclude that the background has a major role in the training process of a CNN. And definitly a random background is really the best as the CNN become robust to any types of background.

8.2 Bias experiments

As we can see on the results, by normalizing the class activation mapping upon computing the metric, we loose information. For the normalized values case, it is really odd for a model with good accuracy and that doesn't look so much outside the leaves to not have the very good bias scores included in the 75% of the distribution. This is because of the way of how a global average pooling layer or a fully connected layer work. They do weighted sums. For instance, let's say we have a kernel with two inputs: -1 and 1. By summing them we get the result: 0. But if normalize the weights of the kernel between 0 and 2 we get 2. This is dead simple but is shows us that if we normalize the data we are no longer following the CNN's algorithm from where theses values cam fom: The CNN computes a score, a sum that is composed by negative and positive values. If we normalize them we end up doing a custom activation and we acutally add bias to our results.
Note that all of our convolutionnal layers had a Relu activation so this is by the global average pooling weights that we got theses negatives values on the CAM.

Now that we know that the untouched values are the correct ones, we can work with them.

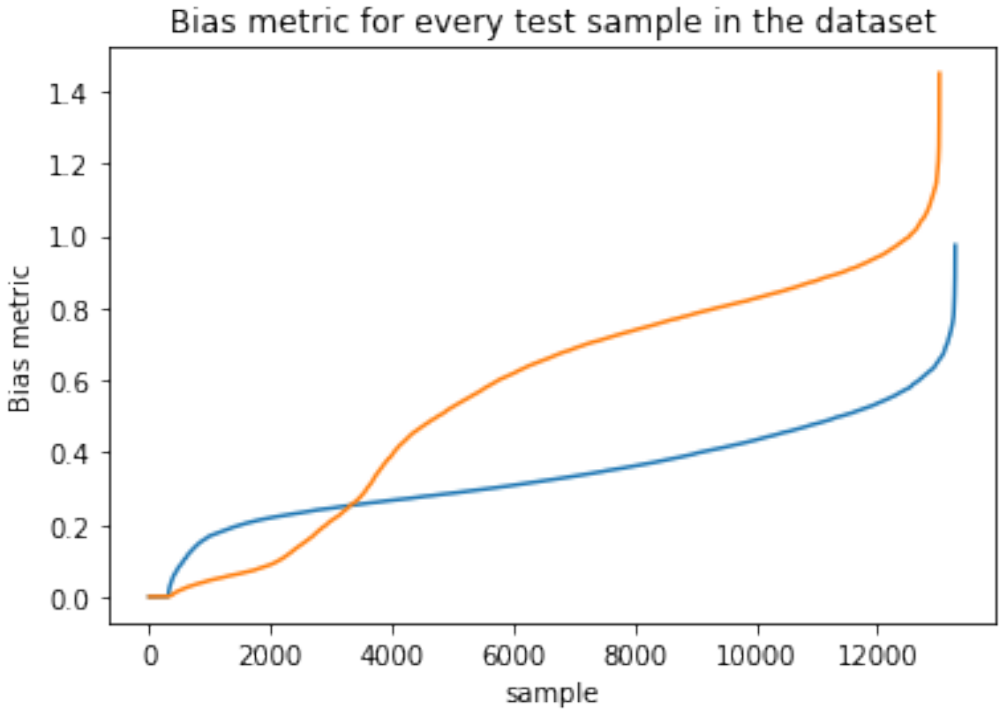

Figure 39: Individually sorted ploting of the bias experiment results (normal training, normal input(blue) and normal training, random input(orange)).

As we can see, by sorting theses values we get a pretty good graphical idea of the performance of the CNN against bias. In blue this is the result of a normal training and a normal inputs and in orange this is the result of a normal training and a random input. As we have see in the previous experiments, normal training handles pretty badly the random inputs and that's what we see on the plot above.

We can also get this bias metric by class:

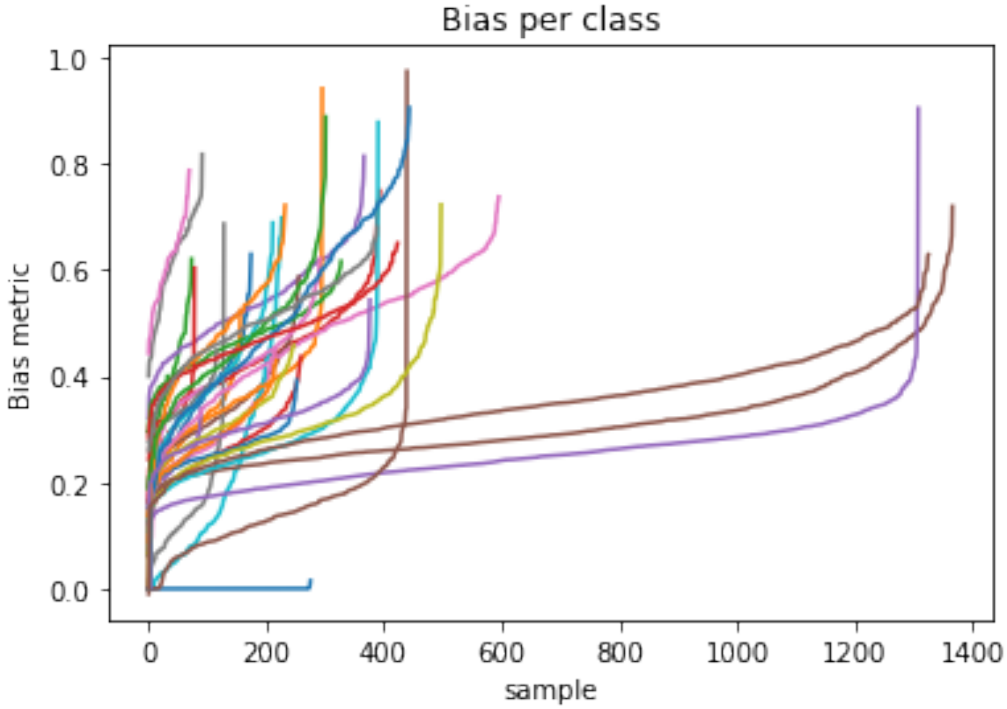

Figure 40: Sorted ploting of the bias experiment results per class (normal training normal input).

Doing so we can get an idea of which class is making the most trouble. And we can also see that the classes that cause the less bias are the ones that have the most samples (except for the first one surprisingly but we know that in the dataset there are samples that cover the entire image so that might explain that).

The wrongs predictions are also very interesting. If we plot their bias metric for every couple of predicted labels and true labels we see that the CNN have a tendency to look more outside the leaf when asked to draw a CAM for the true label. This shows that the CNN is trying to learn features on the leaf and considers outside part as non important feature.

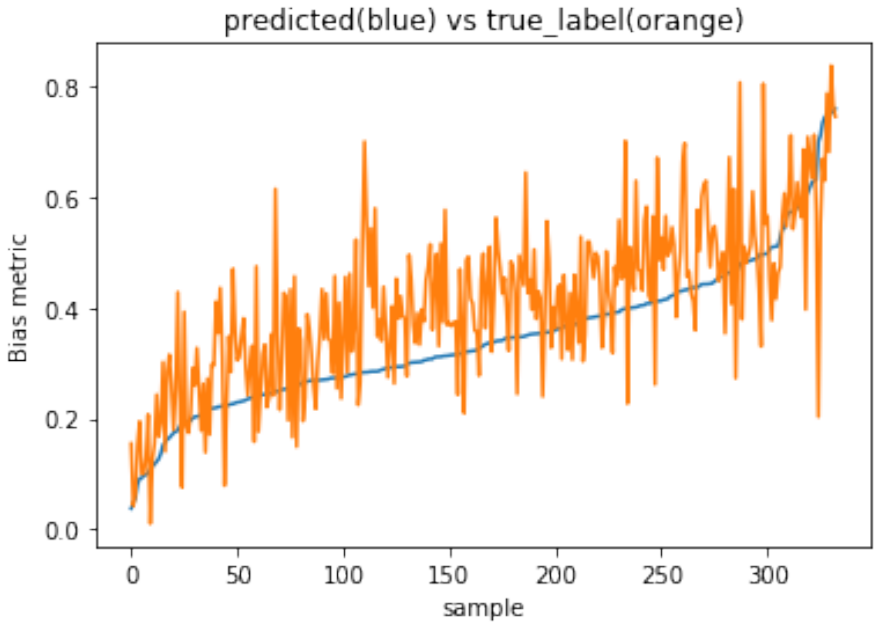

Figure 41: Predicted vs true label (normal training normal input).

9 Conclusion

In this Bachelor work, I have learnt the importance of questioning every choice when it comes to research. This is really important because new results may invalidate previous results. In order to be efficient, it's best to have a modular architecture and to save results between experiments so we don't have to re-run everything if you made a mistake somewhere in the process. So this Bachelor work teached me an other way of working: when we do research, we can't have a time estimation of how long an experiment will take and if it will be worth it. We have to experiment on various data and configurations until we reach wanted results/discoveries.

I'm glad that I've had the occasion to learn more about Tensorflow and Keras. I was interesed in them but never had the time to learn them until this Bachelor work.

10 Known bugs

1. When saving a model with the keras API, the optimizer state is not correctly saved. This forbbids any more training with this model.

11 Future work

Bias metrics results need more exploration as well as class activation mapping data. It would be interesting to use theses values in order to discover new correlations between data sets and CNN performances.

12 References

1. https://plantvillage.org/
2. https://keras.io/
3. https://en.wikipedia.org/wiki/Neuron
4. https://appliedgo.net/perceptron/
5. https://stackoverflow.com/questions/2480650/role-of-bias-in-neural-networks
6. https://drive.switch.ch/index.php/s/5JIWRyeNHQ7IVDg#pdfviewer
7. http://machinelearningmastery.com/gradient-descent-for-machine-learning/
8. https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/
9. https://drive.switch.ch/index.php/s/mXxNTOhbjmE5cJg#pdfviewer
10. https://brohrer.github.io/how_convolutional_neural_networks_work.html
11. https://github.com/fchollet/keras/blob/master/examples/mnist_cnn.py
12. https://keras.io/preprocessing/image/
13. https://github.com/salathegroup/plantvillage_deeplearning_paper_analysis/blob/master/PlantVillageSegme
14. http://math.andrej.com/2010/04/21/random-art-in-python/
15. https://arxiv.org/pdf/1512.04150.pdf
16. https://www.tensorflow.org/
17. http://opencv.org/
18. http://www.vision.caltech.edu/Image_data sets/Caltech101/
19. https://arxiv.org/pdf/1409.1556.pdf
20. https://www.kaggle.com/c/dogs-vs-cats/data

13 Annexes

13.1 Environment installation guide

This guide covers the installation on a amazone g2.2xlarge Ubuntu 14.04 instance.

The repository is divided into two main folder: the "keras" folder is for training models and running experiments ans the "notebook" is for analysing. Now, in order to setup the environment, type the following commands:

```
ssh -X -i <path to key> ubuntu@<public ipv4 adress>
sudo -i
cd /mnt
apt update && apt upgrade -y
apt install git -y
git clone https://github.com/ioannisNoukakis/Bachelor-2017.git
cd /Bachelor-2017/keras
cp deploy_part1.sh /mnt/deploy_part1.sh
cp deploy_part2.sh /home/ubuntu/deploy_part2.sh
chmod +x /mnt/deploy_part1.sh
chmod +x /home/ubuntu/deploy_part2.sh
/mnt/deploy_part1.sh
```

Now go to https://developer.nvidia.com/cudnn and download cuDNN v5.1 for CUDA 8.0 into /home-/ubuntu. You'll have to create an account at nvidia's site in order to make this download. Once it's done run the following commands:

```
cd /home/ubuntu
./deploy_part2.sh
chown -R ubuntu /mnt
```

Finally, add these lines at the end of your .bashrc file that is located under /home/ubuntu

```
export CUDA_HOME=/usr/local/cuda
export LD_LIBRARY_PATH=/usr/local/cuda/lib64/
```

13.2 Command line tool user guide

Into the "keras" directory of the repository you can use the *python3 main.py [args]* command in order to run the experiments described in this Bachelor work with these arguments (each argument is separated by a comma):

- *0* : generates the art background data set and the random background data set.
- *1 [args]* : Fine-tunes VGG16 with the following arguments: *data set's path, path to save the trained model, 'GAP_CAM' for a model with a global average pooling layer or 'DENSE' of a fully connected layer, number of epochs, max images that the program can load into memory (recommended 5'000), '1' to force image resizing to a 256 x 256 shape. '0' to leave images unchanged.*
- *2 [args]* : Generates heatmaps for each sample in the test data set with the following arguments: *input data set's path, model's path, path to save heatmaps, '1' for generate heatmaps for all classes '0' for only the predicted class, how much images it can process at a time (recommended 10), mode for heatmaps generation, 'cv2' or 'tf' (recommened 'tf').*
- *3 [args]* : Computes the bias metric with the following arguments: *path to the heatmaps, number of thread to use.*
- *4 [args]* : Prints how much sample has a class in a data set with the following arguments: *data set's path, 'csv' for csv printing or 'tab' for python array printing or '-' for normal printing.*
- *5 [args]* : Trains the first model with the following arguments:*data set's path, max images that the program can load into memory (recommended 5'000).*
- *6 [args]* : Computes and colors a CAM for each class of a datset. Does not require the command *2* to be run. Uses the following arguments: *model's path, path to save the generated image*
- *7 [args]* : Saves the results of the bias experiment into two files (that are serialized numpy arrays) with the following arguments: *path to heatmaps folder, path to save the correct predicitons, path to save the wrongs predictions.*

The trained models and the bias metric results are to be recovered for local analysis with the Python notebooks provided in the repository.