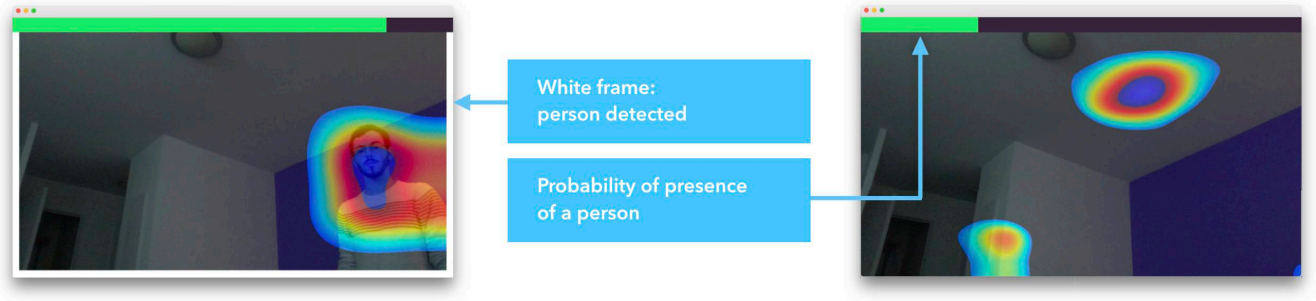
Live Person Detection




```
python3 webcam_cam.py --model ./saved_model/mobilenet_with_gi_data.h5
```

> **Requires:**
> Tensorflow version >= 1.7
> Keras version >= 2.1
> OpenCV version >= 3.3

I. Definition

Project Overview

Neural Networks are often described as black boxes. In this project, however, we will use a method that is based on the interpretation of the internal parameters of a neural network. This method, known as GAP localisation (Global Average Pooling localisation), was introduced in the following paper: Learning Deep Features for Discriminative Localization.

The originality of GAP localisation, compared to other successful localisation methods like Overfeat, YOLO or R-CNN, is the simplicity of the network it uses and of the training it requires.

Using GAP localisation, we will implement an algorithm capable of:

- Detecting the presence of humans in a video
- Identifying "regions of interest" where the humans detected are most likely to be situated

This has multiple applications: surveillance cameras, human-machine interaction, pedestrian detection, automatic focus in digital cameras etc.
In all these applications, we want the method to be fast enough to process a live video.

Problem Statement

Let's define our problem in machine learning terms.

This is a supervised learning problem. Our model will take as an input an image and should output both:

- a category encoded by 0 or 1 (1 if at least one person is present in the image 0 if there are none)
- a heat-map indicating the "regions of interest"

We will see that, although we expect some localisation information in the output of the GAP model, the model will only be trained to solve a classification task.
During the use phase of the model, the localisation information will be extracted from the internal activations of the classification model.

Metrics

We will use metrics commonly used to evaluate the performance of a classification model:

- Accuracy (number of correct predictions / total number of predictions).
- Computation time (in our case, this will be the time needed to output both a category and a heat-map)

Why chose accuracy? Accuracy is a bad measure for classification problems *except* when the target classes in the data are well balanced. This is easy to ensure in the case of binary classification.

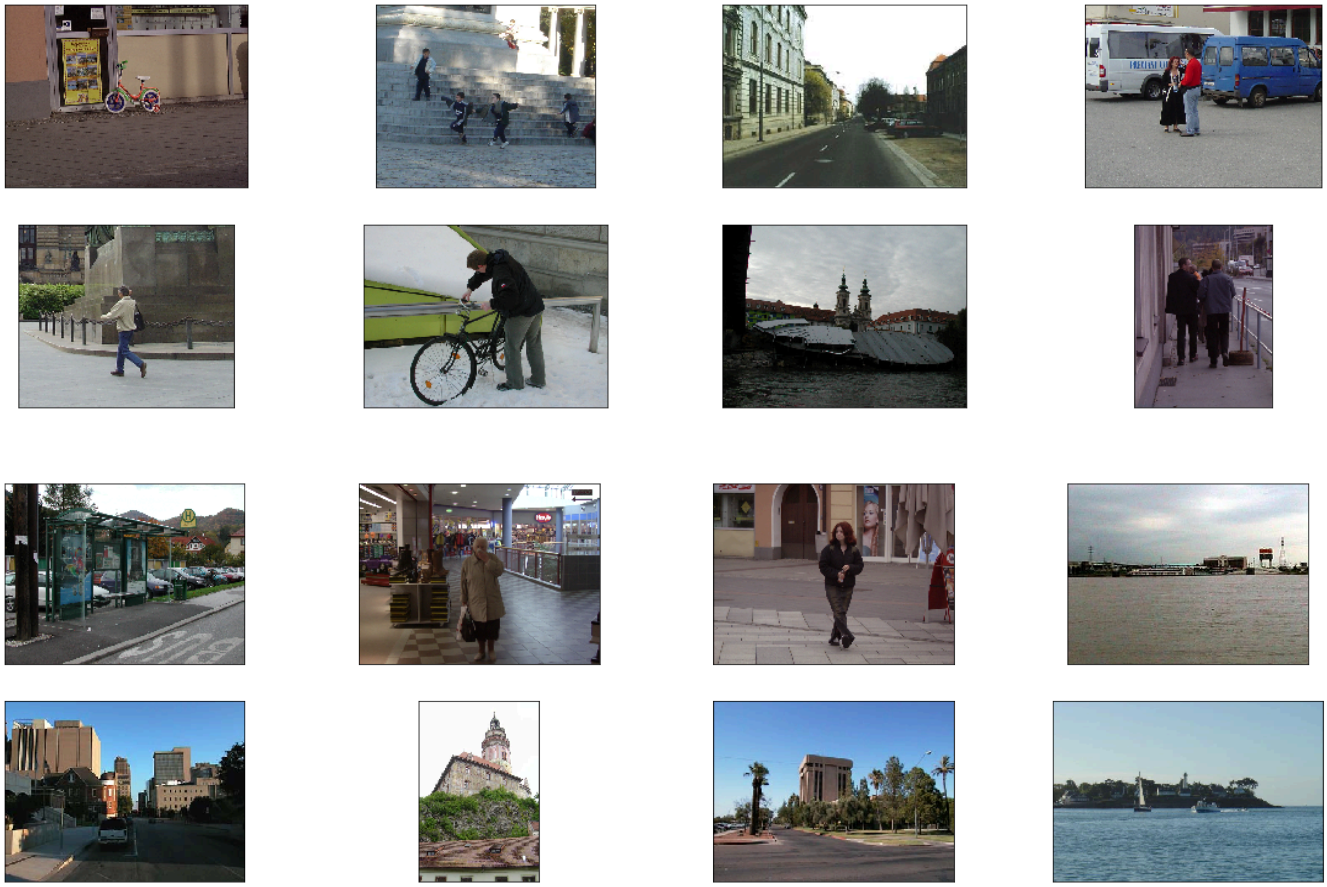
Most of the parameter tuning effort went into the choice of the best number of layers of the convolutional network to unfreeze and of the pertained model.

Since the data is pretty similar to that with which the pertained networks

II. Analysis

Data Exploration and visualisation

The data used to train the classifier comes from the INRIA person dataset.



This dataset contains images with persons (positive) and without persons (negative).

Most of the photos in this dataset are outdoors urban photos and the photos containing people are never close-ups or portraits. We can already expect a model trained on this dataset to be limited because of the lack of variance. Still, this data-set will allow us to prove the effectiveness of the method used in restricted conditions.

The images have different shape and sizes, but all will be resized as 224, 224 images.

Originally, the dataset is unbalanced: there are 1669 negatives and 900 positives which means that a classifier can reach an accuracy of 70% by always predicting negatives. The excess of negatives is therefore not loaded.

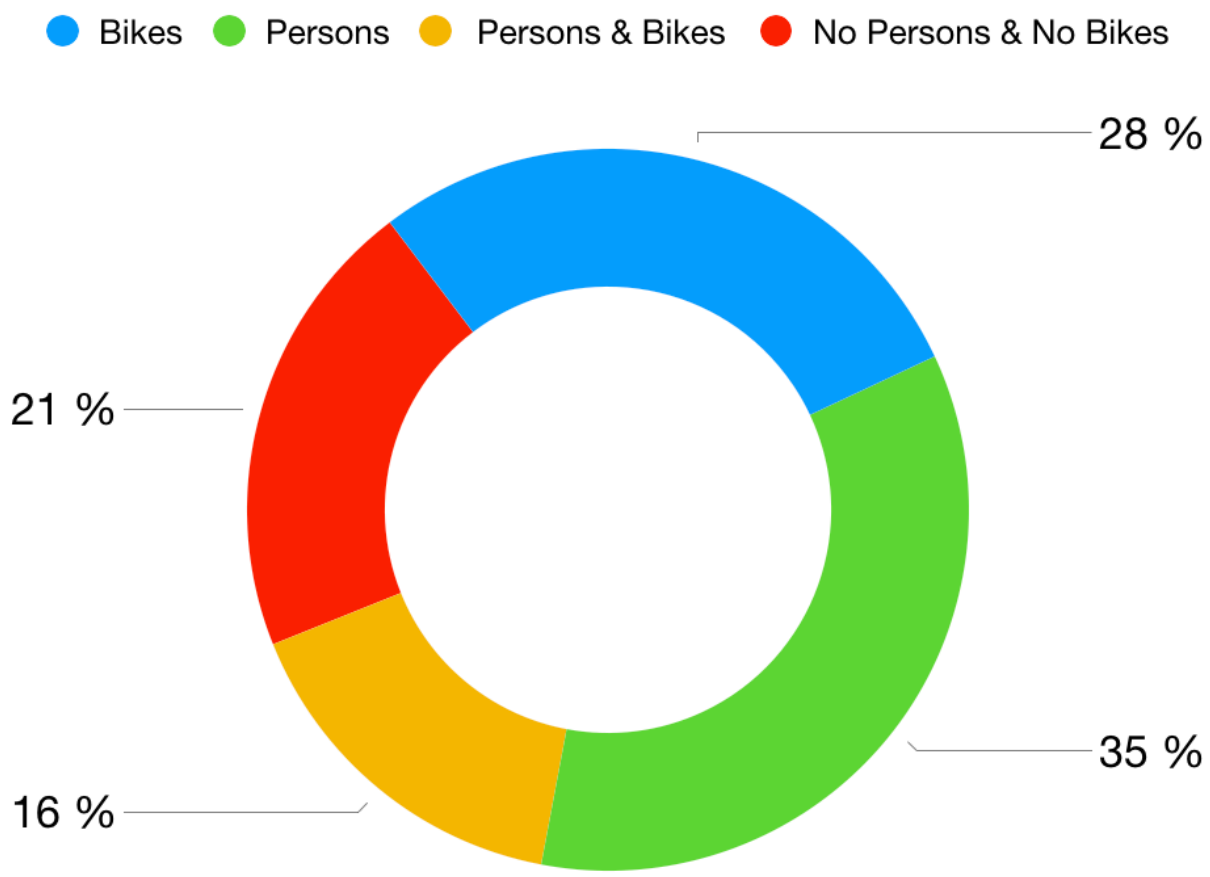
The dataset also doesn't contain a validation set: 20% of the training data is loaded as validation data.

After these two last steps we have the following distribution:

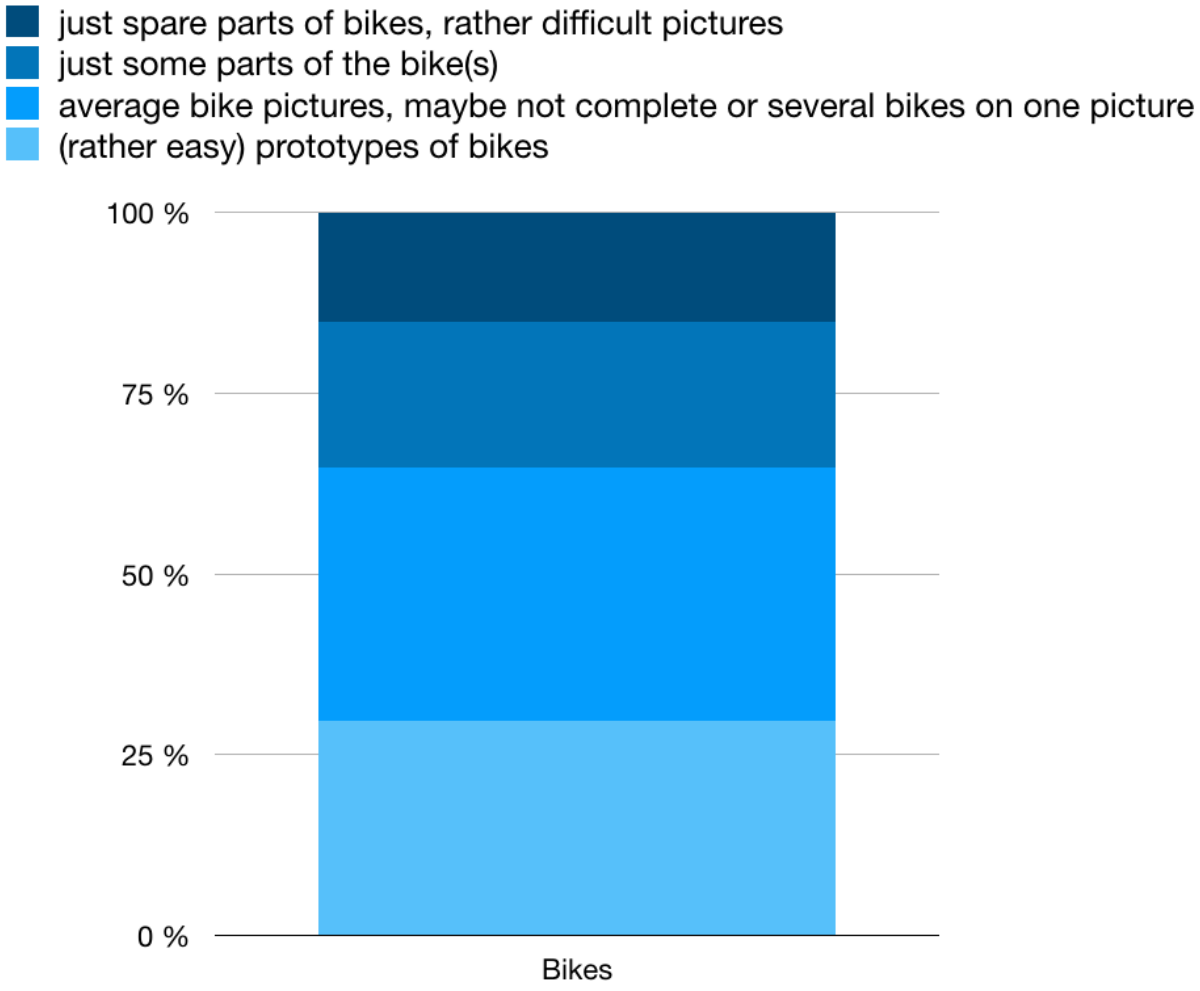
Classes	neg	pos	Total
Train	435	423	858
Valid	184	189	373
Test	227	288	515
	846	900	1746

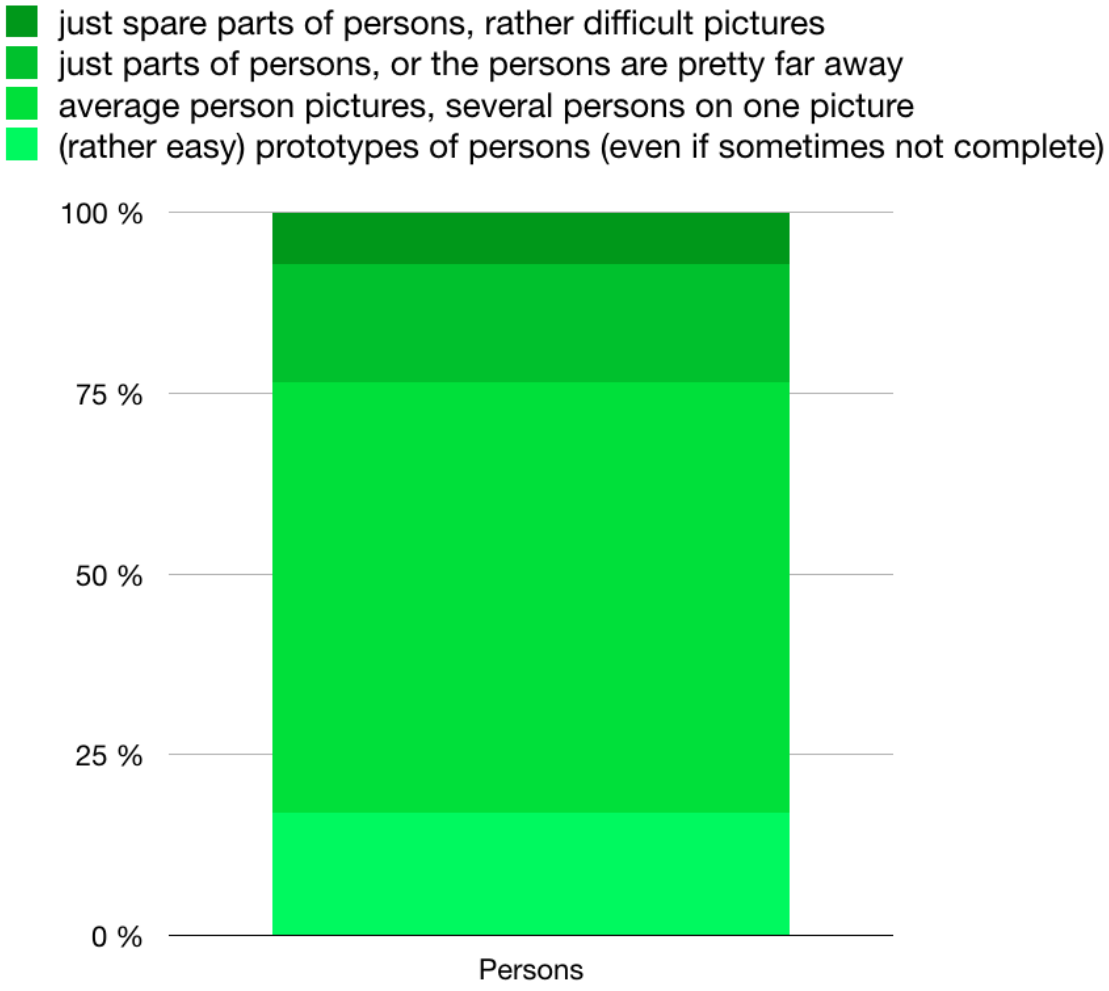
All in all, this dataset is quite small. To mitigate this limitation we use data augmentation (width and height shift, shear, zoom, horizontal flip).

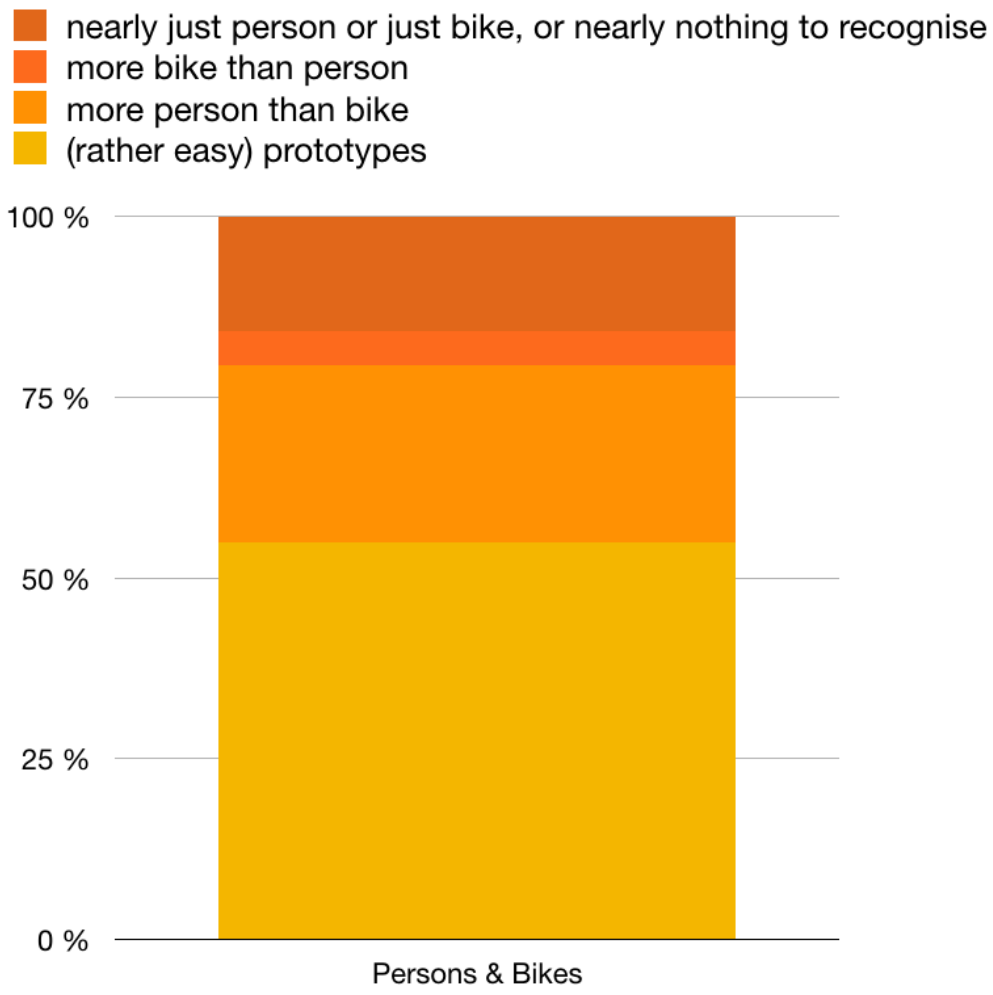
Another interesting limitation of this dataset is that most of the photos are of 4 types of which this the repartition:

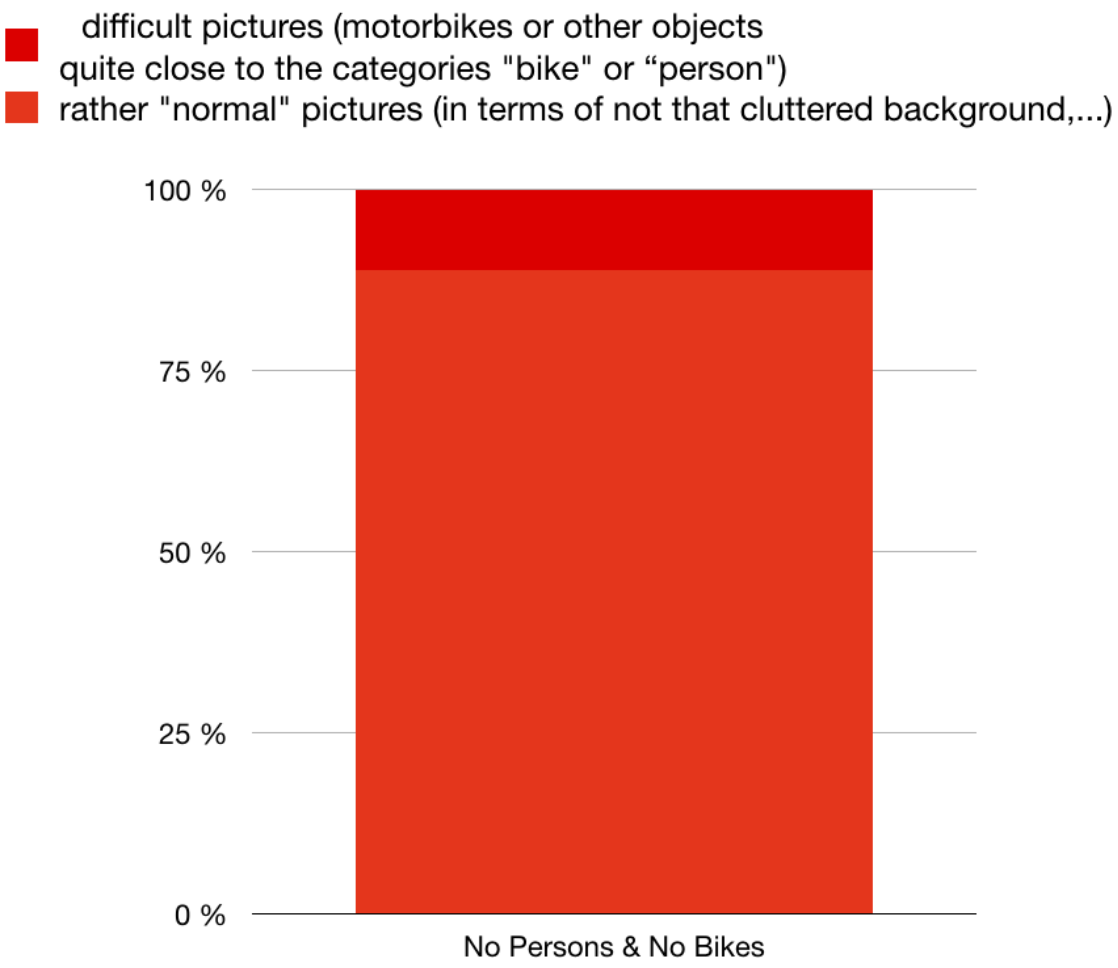


Within each group can be distinguished different categories:


just spare parts of bikes, rather difficult pictures
just some parts of the bike(s)
average bike pictures, maybe not complete or several bikes on one picture
(rather easy) prototypes of bikes
100 %
75 %
50 %
25 %
0 %
Bikes


just spare parts of persons, rather difficult pictures
just parts of persons, or the persons are pretty far away
average person pictures, several persons on one picture
(rather easy) prototypes of persons (even if sometimes not complete)
100 %
75 %
50 %
25 %
0 %
Persons


nearly just person or just bike, or nearly nothing to recognise
more bike than person
more person than bike
(rather easy) prototypes
100 %
75 %
50 %
25 %
0 %
Persons & Bikes



Algorithm and Technique

The originality of the technique we use comes from the fact that:

- it only uses one convolutional network, with a fixed input size
- the model only needs to be trained as a classifier
- the dataset doesn't need to contain "bounding box" information (only labels are required)

To introduce this method, let's start with a quick explanation of what dense/convolutional networks are.

Dense networks

Artificial neurone: A network made up of only 1 neurone can be understood as a logistic regression. It takes as an input a vector $X = (x_1, ..., x_n)$ and outputs

$$y = f(\sum x_i w_i - b)$$

where (w_i), b, known as the *weights* and *bias*, are the parameters of the neurone and where f is its *activation function*. Depending on the activation function, the output is a real number between -1 and 1 or between 0 and 1. Activation functions are more or less smooth versions of the threshold function.

Just as is the case for a logistic regression, the training step consist in adjusting the parameters of the network to minimise an error function, also called the *loss function*, that evaluated the difference between the output computed by the neurone and the expected output.

Multi-layered dense network: Dense neural networks are organised in layers. In a given layer, all the neurones have the same input vector. The first layer takes as an input the input of the network. Each one of the following layers take as an input the outputs of its previous layer.

In a binary classification problem, the network should have a unique output (eg. 0 encodes label 0, 1 encodes label 1). Therefore, the last layer of the network should have one neurone. When there are $k > 2$ labels, the network should have k neurones in the last layer, i.e. one per labels. The output of the network should then be one-hot encoded ($[1, 0, 0, 0, ...]$ encodes label 0, $[0, 1, 0, 0, ...]$ encodes label 1, etc.). (We will see, however, that our network does not abide by this rule: despite the fact that the problem we tackle is a binary classification problem, the GAP method requires 2 neurones in the last layer.)

The choice of the number of layers and of neurones per (non-final) layer depends on the problem and can be optimised using heuristics or by testing multiple architectures.

Limitations of dense networks for image data:

- ** Non-spatial data**: Dense network take as an input a vector. Therefore, an image must be *flattened* in order to be compatible with the input format, which removes the spatial data contained in the image.
- **Redundancy of the parameters:** We expect a pattern to be to be recognised whether it is at the bottom left or at the top right corner of the image. In other words, the parameters should be shared: the parameters responsible for the recognition of a pattern bottom left corner to be the same as those responsible

for the recognition of that pattern at the top right corner. This is not the case in dense networks.

- **Number of parameters:** Because dense networks are **fully connected** (every input gets its own parameter), even small networks require lots of parameters and are therefore very long to train.

Convolutional networks (or ConvNet or CNN)

ConvNets are an answer to the limitation of dense networks.

ConvNets preserve spatial data:

They handle *tensors*, which are arrays of dimension (sample_size x depth x height x width) where:

- in the first layer, the depth is the number of colours of the input image and in the following layers, the number of activation maps (which we define below)
- in the first layer, the height and width are the dimensions of the input image and in the following layers, the dimension of the activation maps

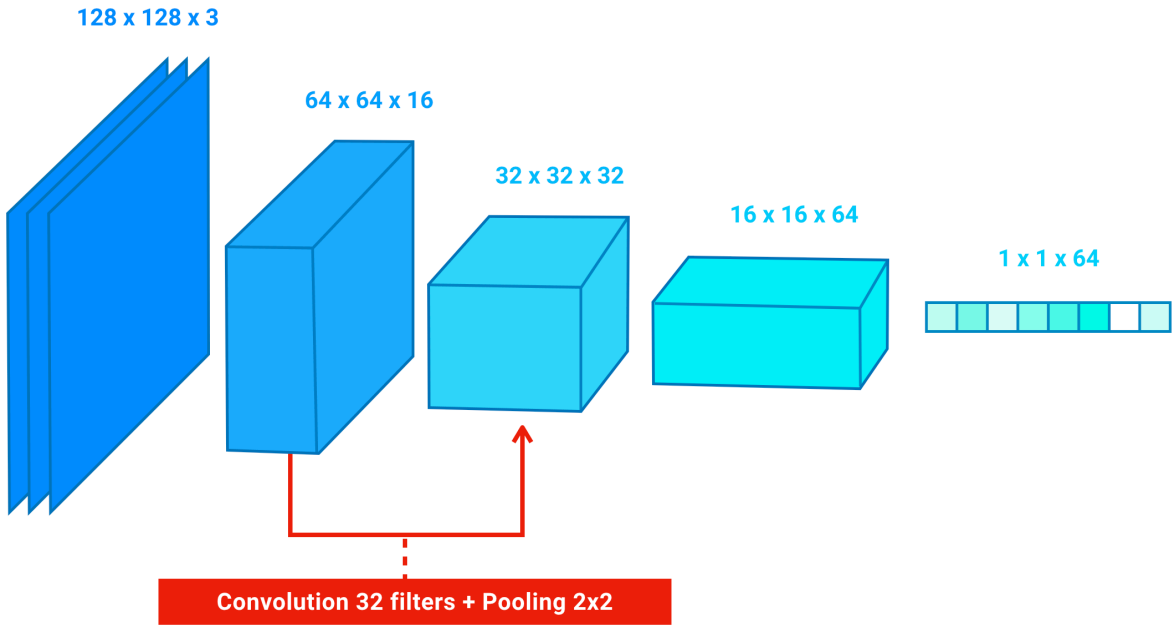
ConvNets are organised in layers which are only *partially* connected. Those layers are of two types:

- **Convolutional layers.** These layers apply multiple convolutional operations to the input image. The parameters are the layers are the coefficients of the convolutions' kernels. The outputs of the convolutions are called *activation maps*.
- **Pooling layers.** There are two kinds of pooling layers:
    - *Max Pooling* of size $i \times j$: the activation map is split in a grid where each cell is of size $i \times j$. For each input cell, the associated output is the maximum value in the cell.
    - *Average polling* of size $i \times j$: For each input cell, the associated output is the mean of the cell.
    - When the pooling size is equal to the size of the activation map, the pooling operation are referred to as *Global Max Pooling* et *Global Average Pooling* (their output are consequently of size 1×1)

Effect of the layers on the dimension of the tensors:

- A convolutional layer that holds n filters has a depth (number of activation maps) of n
- A pooling layer of size $i \times j$ shrinks the dimension of the activation maps by a factor $i \times j$ before it passes them on as an input to the following convolutional layer

The final output of the convolutional layer is a flat vector. This vector is given as an input to a dense network responsible to classify it.



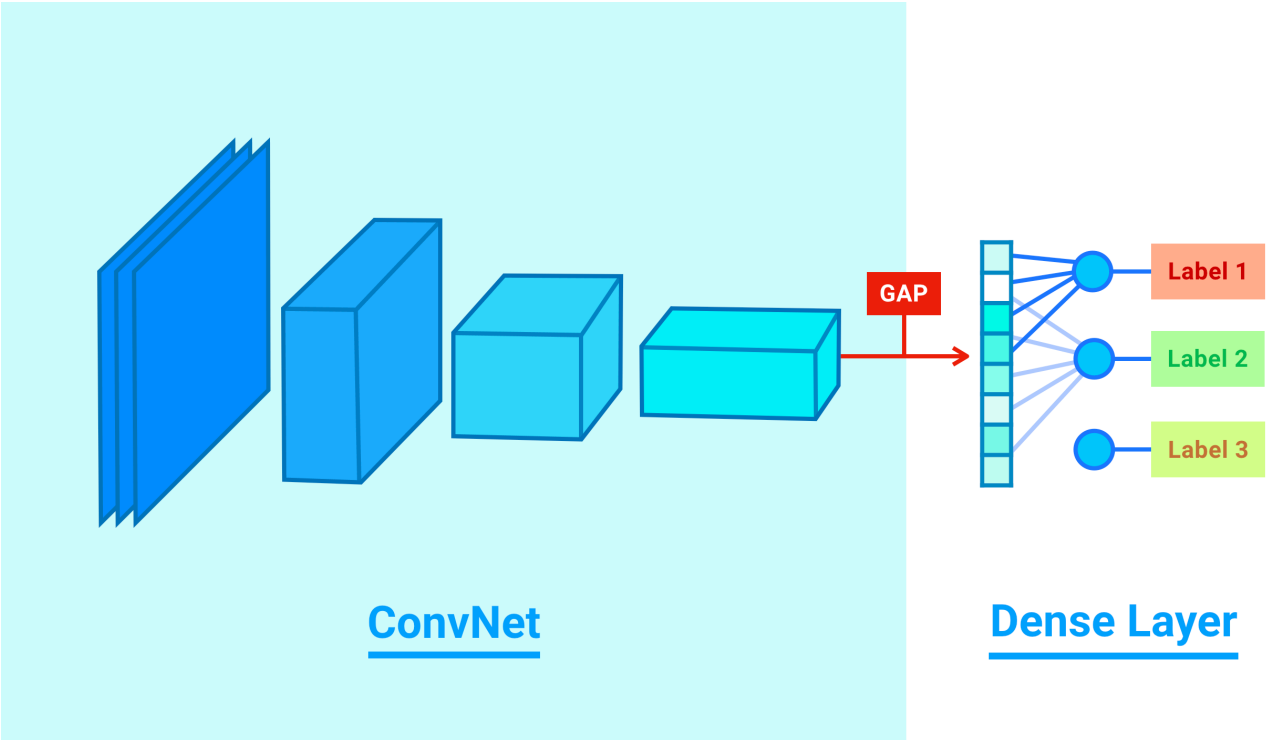
Global Average Pooling localisation

The method used to solve the detection and localisation problem, GAP localisation, is described in the following paper: Learning Deep Features for Discriminative Localization.

Here is a short explanation:

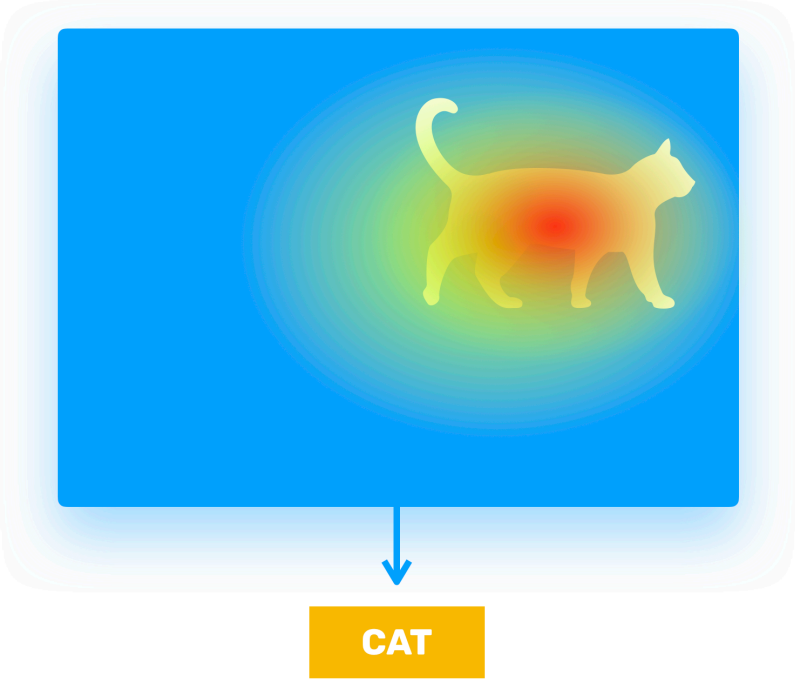
According to the authors of the papers the following convolutional neural, when trained only as a classifier, can be used to localise the classes it classifies. It, therefore, doesn't need bounding box during its training but only labels of the different classes.

The architecture of this neural network is the following:
A convolutional neural network where the last layer is flattened by a Global Average Pooling layer (GAP). Followed by a 1 layer dense network containing as many neurones as there are classes.

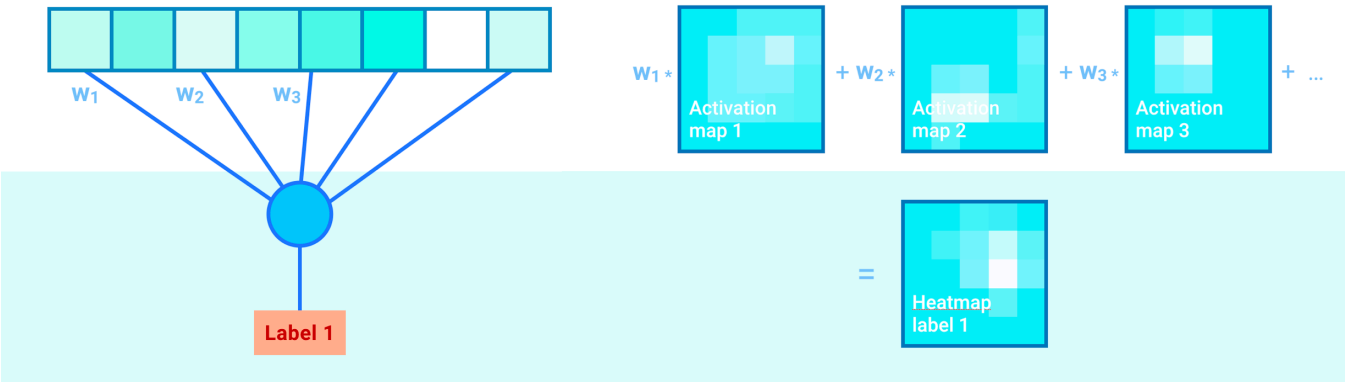


In our case there are 2 classes, therefore 2 neurones in the last layer.

To extract the localisation of the detected classes, the authors propose to extract the Class Activation Maps (CAM). These maps are heat-maps where the hot regions are the regions used by the ConvNet to evaluate a class probability given the input image.



The CAMs can be computed using the activation maps of the last convolutional layer (just before the GAP layer). Because each neurone corresponds to a class, it is possible to weigh each of the activation maps with the weights of the neurones. The computation is the following:



Intuition: If the weight w3 of the neurone associated to the label 1 is large, this means that the 3rd pixel (obtained by computing the GAP of the 3rd activation map) is activated in the presence of an object of class 1. Since the activation maps contain spatial data, each pixel corresponds to a region of the input image.

Benchmark

Accuracy:

An accuracy above 95% is considered to be very good for a binary classifier (dog/cat classification)

The results will hopefully prove that imposing a GAP architecture doesn't harm the classification results.

Computation time:

The detection should be fast enough to process a live video. For the output detection video to be smooth we need it to be able to process at least 5 images per second.

III. Methodology

Data Preprocessing

Loading the data

- ... transform each image into a normalised 4D tensor of shape (1, 224, 224, 3) suitable for supplying to a Keras CNN
- ... aggregates all the tensor of the sample into a larger 4D tensor of shape (samples_size, 224, 224, 3)
- ... transforms class labels into one hot encoded labels.
- ... creates a validation folder using 20% of the original training data.
- ... balances the data.

Implementation

Development environment

Training neural nets on a laptop is a lost cause. One training epoch on a 50 layers neural network (ResNet50) with data augmentation takes more than 45 minutes.

The following Google Cloud compute engine configuration was used for this project:

- 6 CPUs, 32 GB memory
- GPU NVIDIA Tesla K80
- CUDA toolkit and cuDNN installed to ensure that the GPU is used for the computation

With this architecture, a training epoch can be completed in 3 min.

The algorithm was implemented using Python 3 and the Keras and Tensorflow libraries.

Transfert learning

In 2017, the UC Berkeley team composed of Yang You, Zhao Zhang, James Demmel, Kurt Keutzer boasted:

> “We finish the 100-epoch ImageNet training with AlexNet in 24 minutes, which is the

> world record. Same as Facebook's result, we finish the 90-epoch ImageNet training with ResNet-50 in one hour. However, our hardware budget is only 1.2 million USD, which is 3.4 times lower than Facebook's 4.1 million USD."

Training a neural network from scratch is therefore not accessible to individuals. A solution is to reuse a pre-trained network.

Architecture and training configuration

In this project, we used nets pre-trained on ImageNet (provided by the Keras library). All the configurations of the network used are listed below (the exact implementation can be found in `utilities/classifier.py`):

Architecture:

- Convolutional layers of the pre-trained network followed by a GAP layer and a dense layer
- Only the 2 last convolutional layers are unfrozen (because the dataset is small)

Training configuration:

- ReLU activation functions in the hidden layers to avoid the 'vanishing gradient'
- Optimisation algorithm: adam (step: 0.001)
- L2 regularisation (0.1) to avoid overfitting
- Batch-Normalisation to avoid overfitting
- Data augmentation to avoid overfitting
- Batch-size: 32
- Cross-Validation: only the model with the lowest validation loss is saved

Generation of the CAM images

This part is the most tricky since it is not implemented in the Keras library. It is also easy to get wrong: all the implementation that can be found online created a new Keras `Model` (taking as an input an image and outputting a CAM) at each time a CAM needs to be generated. This is very inefficient. In this implementation, redundant calls are eliminated: only one CAM-generating model is created.

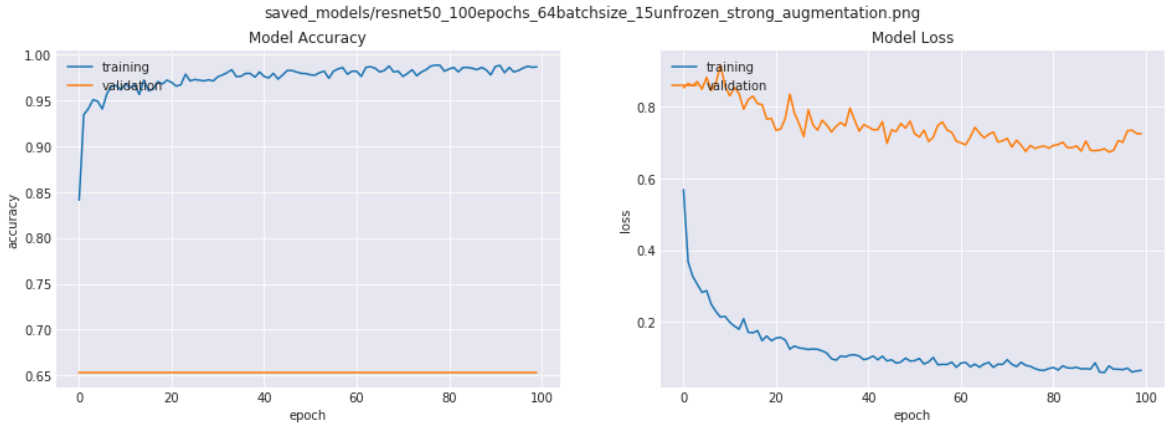
The complete commented implementation can be found in `utilities/classifier.py` in the

`Classifier.cam()` method. Here is an excerpt:

```
# Initialize with the cam with right shape
cam = np.zeros(dtype = np.float32, shape = conv_outputs.shape[0:2])

# Weighted sum of the cam = \Sum_i (cam_i * weight_i)
for i, w in enumerate(class_weights[:, class_number]):
    cam += w * conv_outputs[:, :, i]

# Normalise and resize the cam
cam /= np.max(cam)

# Resize the cam
cam  = cv2.resize(cam, (height, width), interpolation = cv2.INTER_CUBIC)
```

Refinement and parameter tuning

- Most of the parameter tuning effort went into the choice of the best number of unfrozen layers in convolutional network.
  Since the data is pretty similar to that with which the pre-trained networks where trained, it is not necessary to train many of the last layers. Moreover, because the dataset is small, when too many layers are unfrozen, the model overfits the data.
- Not much hyper-parameter tuning was needed apart from that. The values recommended by Keras (learning rate, intensity of the regularisation for examples) gave the best results.
- It is interesting to note that because of the limited size of the dataset, it is difficult to train a very deep network such as ResNet50.
  Here are the learning curves for this model (evolution of the accuracy and loss during the training iteration, on the training and testing set):



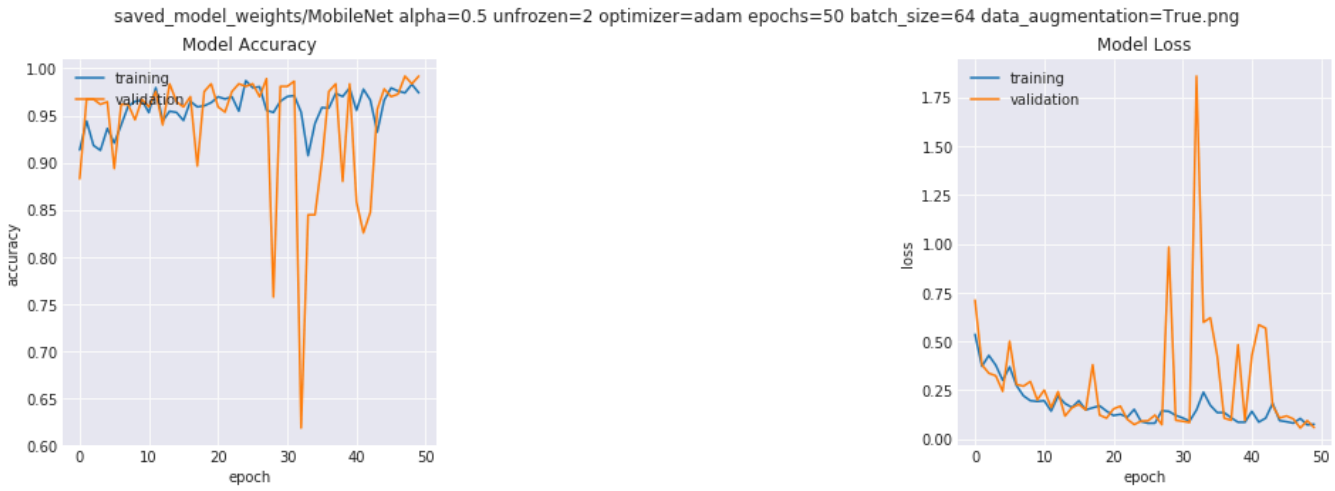
This is a good example of overfitting: after a few iterations, the results become very good on the training set but stay very bad on the validation set.
We, therefore, prefer smaller networks.

- The pre-trained model used in this project is MobilNet. This architecture was introduced in 2017, therefore after the Learning Deep Features for Discriminative Localization paper (2015) and is more performant (similar accuracy but smaller computation cost) than the architectures proposed in the paper.

IV. Results

Model Evaluation and Validation

Here are the learning curves:



The learning curves show a good convergence in less than 50 epochs. During all the training iterations, and except for some spikes here and there, the results are similar on the training and validation sets. This means that this network didn't suffer from overfitting.

The model doesn't underfit either: in less than 50 epochs, the MobilNet gives an **accuracy of 97.3282% and a loss of 5.20%** on the **test set**.

More interesting, the creation of a CAM from an input image (prediction of the class + generation of the heat-map) takes less than **0.1 second on a laptop**.

Here are some example outputs. The detected class is highlighted with a blue frame:

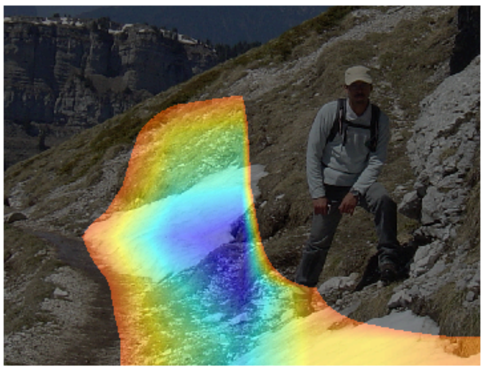
Model=MobileNet alpha=0.5 unfrozen=2 optimizer=adam epochs=20 batch_size=32 data_augmentation=False Prediction=non-human GroundTruth=non-human

non-human cam

human cam

Model=MobileNet alpha=0.5 unfrozen=2 optimizer=adam epochs=20 batch_size=32 data_augmentation=False Prediction=human GroundTruth=human

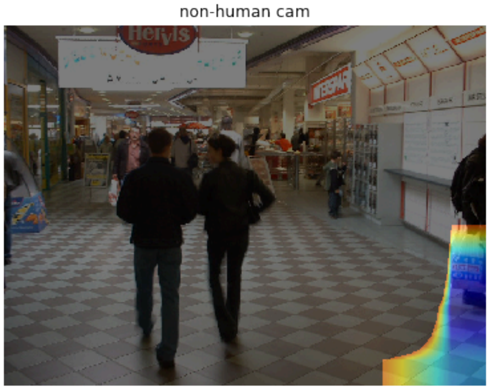
non-human cam



human cam

Model=MobileNet alpha=0.5 unfrozen=2 optimizer=adam epochs=20 batch_size=32 data_augmentation=False Prediction=human GroundTruth=human

non-human cam

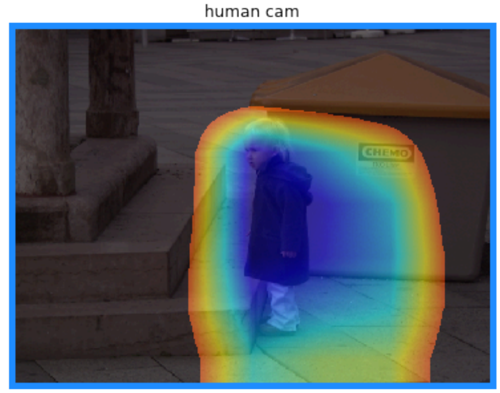


human cam

Model=MobileNet alpha=0.5 unfrozen=2 optimizer=adam epochs=20 batch_size=32 data_augmentation=False Prediction=human GroundTruth=human

non-human cam

human cam



Model=MobileNet alpha=0.5 unfrozen=2 optimizer=adam epochs=20 batch_size=32 data_augmentation=False

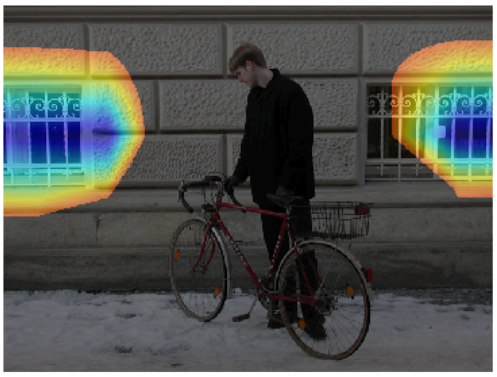
non-human cam

Prediction=human GroundTruth=human

human cam

Model=MobileNet alpha=0.5 unfrozen=2 optimizer=adam epochs=20 batch_size=32 data_augmentation=False

non-human cam

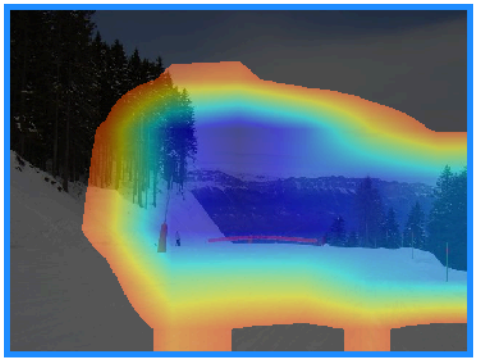


Prediction=human GroundTruth=human

human cam

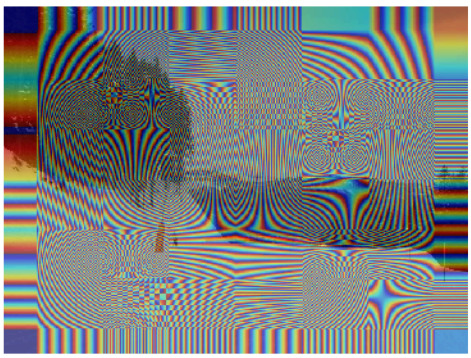
Model=MobileNet alpha=0.5 unfrozen=2 optimizer=adam epochs=20 batch_size=32 data_augmentation=False

non-human cam



Prediction=non-human GroundTruth=non-human

human cam



Observation:

- When no human is detected, the 'negative' heat-map highlights lines, buildings, grids etc.
- When no human is detected, the 'positive' heat-map doesn't highlight anything in particular
- Groups of humans aren't separated by the algorithm

Justification

The final accuracy and computation time results found are stronger than the benchmark.

The localisation results are satisfying for the method proposed in the paper and for the size of the dataset. But there is one major limitation to this method: the network doesn't separate individuals when they are in a group, close to each other.

V. Conclusion

Visualisation

The (summarised) final architecture is the following:

```
Layer (type)                 Output Shape              Param #
=================================================================
input_15 (InputLayer)        (None, 224, 224, 3)       0
_________________________________________________________________
conv1_pad (ZeroPadding2D)    (None, 226, 226, 3)       0
_________________________________________________________________
conv1 (Conv2D)               (None, 112, 112, 16)      432
_________________________________________________________________
conv1_bn (BatchNormalization (None, 112, 112, 16)      64
_________________________________________________________________
conv1_relu (Activation)      (None, 112, 112, 16)      0
_________________________________________________________________
conv_pad_1 (ZeroPadding2D)   (None, 114, 114, 16)      0
_________________________________________________________________
conv_dw_1 (DepthwiseConv2D)  (None, 112, 112, 16)      144
_________________________________________________________________
conv_dw_1_bn (BatchNormaliza (None, 112, 112, 16)      64
_________________________________________________________________
conv_dw_1_relu (Activation)  (None, 112, 112, 16)      0
_________________________________________________________________
conv_pw_1 (Conv2D)           (None, 112, 112, 32)      512
_________________________________________________________________
conv_pw_1_bn (BatchNormaliza (None, 112, 112, 32)      128
_________________________________________________________________
conv_pw_1_relu (Activation)  (None, 112, 112, 32)      0
_________________________________________________________________
...
_________________________________________________________________
conv_pad_13 (ZeroPadding2D)  (None, 9, 9, 512)         0
_________________________________________________________________
conv_dw_13 (DepthwiseConv2D) (None, 7, 7, 512)         4608
_________________________________________________________________
conv_dw_13_bn (BatchNormaliz (None, 7, 7, 512)         2048
_________________________________________________________________
conv_dw_13_relu (Activation) (None, 7, 7, 512)         0
_________________________________________________________________
conv_pw_13 (Conv2D)          (None, 7, 7, 512)         262144
_________________________________________________________________
conv_pw_13_bn (BatchNormaliz (None, 7, 7, 512)         2048
_________________________________________________________________
conv_pw_13_relu (Activation) (None, 7, 7, 512)         0
_________________________________________________________________
global_average_pooling2d_11  (None, 512)               0
_________________________________________________________________
dense_layer (Dense)          (None, 2)                 1026
=================================================================
```

```
Total params: 830,562
Trainable params: 819,618
Non-trainable params: 10,944
_________________________________________________________________
```

Here are some screenshots of an application of the method implemented in this project. This application can be launched quite easily with the following script and is based on saved trained model so doesn't require to download the dataset: `webcam_cam.py` (see also `python3 webcam_cam.py --help`).

The classification and computing time are very good and the live video is fluid. However, the localisation results are limited: groups of people are recognised as one region of interest and close-ups (only a face present on the video) aren't always recognised properly.

Reflection

This results should be put into perspective: the authors of the paper used at least tens of thousands of images per class, when the model presented above only uses 855 images in total in the training set. The results are quite satisfying for such a small data set (mainly thanks to data augmentation).

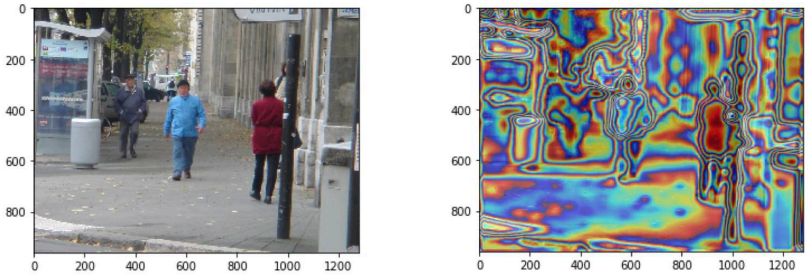
All in all, we have proved that:

- using a network trained only as a classifier, we can obtain localisation information
- the constraints that the GAP method imposes on the architecture (only one dense layer at the end of the network preceded by a GAP layer) don't harm the classification results

Here is a step by step recap of the process I went through to build this project:

- Loading and pre-processing the dataset

- I then started with a simple implementation with a network made from scratch in Keras (no transfer learning). This attempt wasn't successful... Here is an example of the CAM I got with this first method:



- Using transfer learning with a ResNet50 neural network I created a new network. This wasn't successful either...
- I finally tried to implement transfer learning with a smaller network, MobileNet, which gave me the best results.
- The last step was to optimise and factorise the code

But the project took more than 5 neatly independent steps... Here are some of the things I learned during this project:

- Debugging a neural network is HARD: I realised many times after 1 or 2 hours of training a model that I hadn't implemented a feature to save my weights or that I made a small mistake in the training parameters etc. Sometimes my computer will crash after 1 hour or the connection to the cloud computing engine would get interrupted
- Computing on GPU is far more efficient than on CPU. I also learned to install the CUDA toolkit and cuDNN
- I also got to test out both AWS and Google Cloud. Google Cloud seems overall to be a better solution for occasional use and for a project, especially since they offer one year of free usage
- Choosing the right dataset is very important as well as balancing it (in the case of classification problem)
- When manipulating neural network, it is important to take into consideration the complexity of the code (cost in time) and avoiding duplicate operation can be crucial

The two biggest challenges were:

- To get an initial version of the code working!

- To implement a CAM generation algorithm efficient enough to be used in a live video stream. This was also the most interesting part of the project because it forced me to rethink the whole architecture of my code (from a bunch of function to a more efficient Oriented Object implementation)

Improvement

The 2 principal limitations of the solution presented above are the following:

- The resolution of the localisation is limited by the size of the last activation map.
- The training set used in the project is very small, and it is difficult to find a large, balanced dataset corresponding exactly to the problem.

Based on these limitations, the possible improvements are the following:

- Using encoder-decoder networks as described in the following paper Fully Convolutional Networks for Semantic Segmentation
- Using a larger database like the Google Open Dataset