Report ATIA

Domenico Villani

January 2020

1 Assignment 1

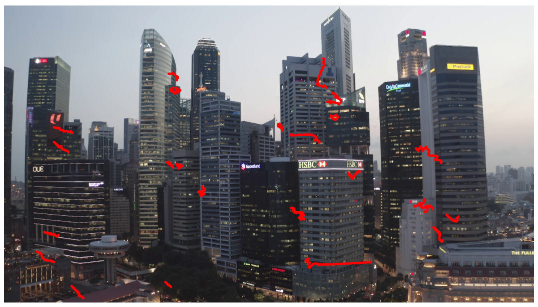
This report describes the KLT tracking algorithm developed during the course in Advanced Topics in Image Analysis at KU.
The main focus is to develop a working prototype that shows the understanding of theory behind this tracking technique and the ability to translate the knowledge in real code. Because of these requirements the usage of code libraries has been limited. The code has been implemented in Python 3. In some cases, such as the convolution, I have implemented and used the functions, but then I have chosen to use the library version to provide a faster code execution.
This report as well as the code provided will be modified taking in consideration the feedback I will receive, in order to overcome the limits of the present implementation. The algorithm can be run executing the python program `Assignment_1.py` (which points to the .mp4 file in the `Ass_img` folder) in the present folder. All the code is in `KLT_traker.py` this is because I want to show all the code in the same file in order to help the debugging, the final implementation should be divided in different folder with dedicated classes.
The Lucas-Kanade algorithm is a sparse optical flow algorithm, meaning that it tracks only some interesting pixels between consecutive frames. These pixels are points that we previously identified as feature points. The features are applications specific, they depends on the accuracy and the type of the results we want, in this case we used the Harris corners detector algorithm to analyse the first frame and find the points which will be tracked.

The Harris corner detector identifies interest points where we can find more than one edge around a pixel. It calculates the image gradient of a pixel respect to the x and y axes, then it computes a tensor T which is the dot product of the vector composed by the two gradients x and y. The eigenvalues of the tensor T gives us the solution, in case they are both large we have a corner, in case only one of them is large we have an edge and if none of them is large we can assume there is no point of interest.

Once we calculated the Harris corners we apply the KLT algorithm on those points. We select a window and for every feature point we calculate the image gradient respect to the x and y axes and then apply a Gaussian kernel, as we did

for the Harris corners, then we stack the gradients in order to form a matrix A which then we multiply by the transposed version of itself to find the structure tensor T that will be multiplied to solve the equation system.
In order to calculate the second part of out equation system e, we need to interpolate the window of the second image the J-patch, then we subtract the J-patch interpolated from the same window position in the first image, this operation gives us the difference D between the two frames. Then we get the dot product between the matrix A and D, we multiply the result by -1 and this gives us the e. Now we can solve the system of equations between the inverse of the tensor T and e. This last operation gives us an approximate new positions of the new feature points because of the Taylor series truncation, so we need to apply the last part again recalculating the e respect to the new points. The iteration is application dependent and in this case I chose to 5 loops for performance reasons.

The code seems to work as described above, despite some points missing the right position after some iterations. I guess the problem arise with the Taylor series truncation, probably it needs more loop iterations. Also the changing light condition which changes the texture of some of the detected corners from frame to frame. I chose to set a threshold for the Harris corners that let identify only 35 points at the beginning, this is because the frames are dense in terms of information and contain many corners, this would make the algorithm run for a very long time trying to recalculate thousands of corners for each new frame. As we can see in the following image (fig. **??**) the algorithm works and the points are tracked correctly, in some cases as I said the points a lost and the corners should be recalculated after some iterations.

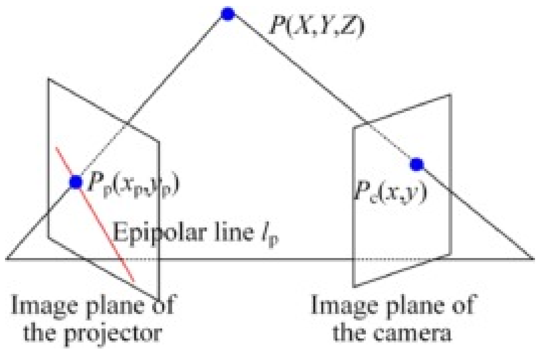


2 Assignment 2

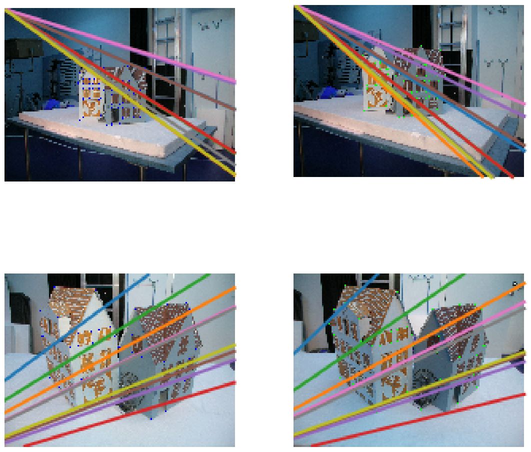
The implementation contains five methods and the code in the main section to execute it. The algorithm can be executed simply running the *Assignment_2.py* file which calls the functions inside the *ransac.py* file. Epipolar Geometry describes the geometric relations between a set of images. Without knowing the camera parameters we can map points from one image to the second image. The fundamental matrix **F** describes this intrinsic geometry. It is a 3x3 matrix of rank 2. If we have two points of view of the same point in the space x_1 and x_2 coming from two different images, they satisfy the equation:

$$x_2^T F x_1 = 0 \tag{1}$$

The epipoles between two points of view is the intersection of the image planes with the base line that connects two camera centers, when a point x is projected into the image plane, the line connecting the epipole with the projected point is the epipolar line. Calculating the epipolar lines of one image respect to the other we can map these points (fig 2).



The **RANSAC** algorithm is an iterative method, which given a measure, is able to identify outliers that do not fit the model among the given points. The *ransac* method randomly select eight different points subset on which it calculates the F matrix. Then evaluates the best F using the *Sampsom distance* and return it. Once the matrix is calculated, it is used to compute the epipole points using the F and its transpose for the right and left epipole, then the lines are drawn. The *compute_F* method is implemented following [1], the points are normilised and the F matrix is calculated following the equation in [1], then the singularity constraint is ensured as exposed by *Hartley and Zisserman*. The *Ransac* algorithm creates a for loop where each time randomly selects 8 points to calculate the F, then the *Sampsom* distance is applied to select the best F. I fixed the *Sampsom* distance, now it works as expected, the result of the matrix calculation are still not optimal and they can be improved.



3 Assignment 3

The last assignment is meant to explore the possibilities of object recognition, in particular image classification of leafs from the **PlantCLEF 2015** competition dataset. The the category selected is the LeafScan which contains 351 different classes of leaves. In order to limit the computational burden it was suggested to focus on the classes containing the most numerous amount of samples. I chose to select two sub-samples one containing the first 20 classes and one containing the first 30 classes. After this selection the number of samples remaining is rather small (6058 for the 30 class subset), as suggested I chose to use transfer learning in order to avoid overfitting the network on this small dataset. Transfer learning is a popular technique that allows to reuse a pretrained model as starting point for our task. Using a pretrained model we can take advantage of the convolutional filters which are already trained on a huge amount of samples. It can be used for different purposes such as feature extraction where the pre-trained model is used to pre-process images and extract relevant features or, as in this case, it can be integrated in a new model where the pretrained weights are static, not changing during the training, and only the rest of the model changes its weights. A Convolutional Layer is a set of filters that is applied to an input vector (the image) which can be composed also by a set of images, a Batch. In this case the Batch is set to 32. Sending multiple images at a time helps the to speedup the training, taking advantage of the GPUs parallel capabilities. At the beginning of a training the filters values are randomly initialized, then during the training they change using backpropagation. At the end of a training their values represents different features learned by the images contained in the training dataset (fig 1). The first layers represents low level features such as lines or corners, then going deeper into the network we find higher level of abstraction. During prediction, when an image is sent into the network, will

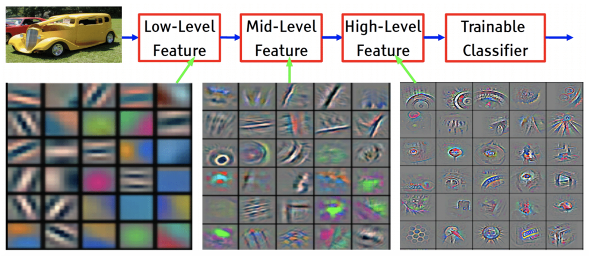


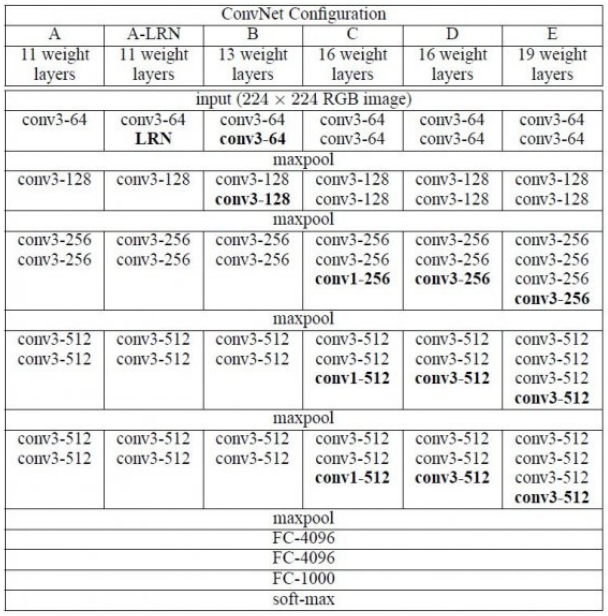
Figure 1: The VGG16 filters after training

activate some of these filters which are more sensitive to the specific feature of the image.

A common method used in transfer learning where the model is a **Convolutional Neural Network** is to froze the weights in the convolutional layers and redesign the fully connected part of the model in order to fit the specific task. This method allows us to take advantage of the pretrained filters trained on the **ImageNet** dataset which contains over 15 million high-resolution images from around 22000 categories. This technique gives us the possibility of using already optimized filters for the image features that are common also to our dataset, then training only the last layers results in a sort of fine-tuning the network for our specific task.

In this case I used the **VGG16** (fig 2) model which is a **Convolutional Neural Network** architecture composed by 16 layers, 13 convolutional and 3 fully connected and 4 pooling layers. The model uses mainly 3x3 filter convolution, in some cases a 1x1 filter which is linear transformation of the input channels with the activation function which add the non-linearity. I redesigned the last 3 fully connected layers, they still use 4096 neurons in the 2 layers after the convolution, but the last layer is respectively 20 and 30 neurons depending on how many output classes we need. The model contains a total of 134,342,484 parameters for the 20 class and 138,357,544 for the 30 class architecture, as we used transfer learning only the last layers are trainable and the rest of the weight are frozen, for this reason only 119,627,796 parameters are trainable for the 20 class and 119,668,766 parameters for the 30 class network.

All the hidden layers use **ReLU** as activation function which adds the non-linearity to the network, in fact without an activation function the network would be a summation of different linear systems, which is in itself one big linear system. The last layer applies the **Softmax** activation function, it is commonly used in a classification problem with $K > 2$ classes because it is able to output a probability measure out of each class, usually the class with the highest probability is selected, but in some cases a threshold can be set in order to enforce the confidence of our prediction. VGG16 uses 5 **MaxPooling** layers with a 2x2 pixel window and stride 2, they are layers without any parameter, then not trainable, which are used to downsample the image taking the maximum among the pixels window. This operation helps to create higher level of



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 2: The VGG16 architecture

abstraction going deeper in the network and creates a sort of multi-resolution filter bank[2]. The loss function used is the multiclass cross-entropy. The optimization algorithm is the **RMSProp** which takes in consideration not only the last gradient, but also the precedent in order to adapt the learning rate during the training, in this way we can avoid getting stuck in a local minima without setting the learning too small, resulting in a very slow training process. During the development I found the amount of testing samples too small, they were not covering all the classes after the first 30/20 classes where selected. For this reason I tried to split the dataset that was supposed for the training/validation in 3 different sets, training, validation and testing. In this way I keep 928 and 1212 samples for testing for the 20 and 30 most numerous classes.

The training loss and accuracy for the training and validation sets can be seen in fig. 3 and 4. The model evaluation for the fifth epoch (which seems to be one of the best) has a loss of 0.18 and the 95.36% of accuracy.

As we can see from the plots after few iterations (the fifth) the model start to be stable, then it start to flip up and down in terms of loss and accuracy, this is a sign of overfitting and we should not train it over fifth epoch. The same is valid for the 30 classes case, where the best model seems to be the one after four iterations which gives us a loss of 0.2569 and a 92.73% of accuracy.

Before being able to train I created a **PlantData_ranked.csv** file containing only the **LeafScan** category data. Then the csv file is used to select the most numerous classes. I used **Google Colab** for training.

It was suggested to use the metrics from the **PlantCLEF 2015** competi-

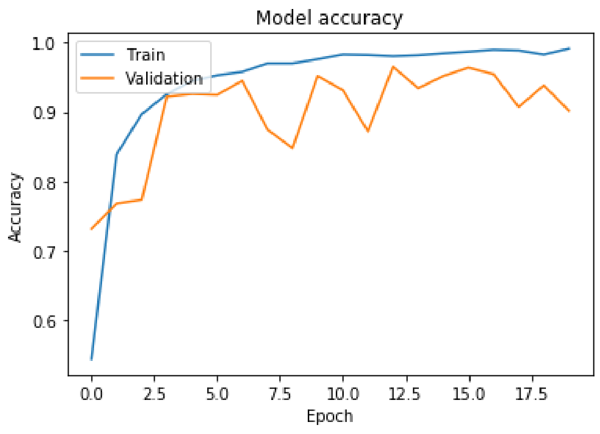


Figure 3: Accuracy 20 sample classes over 20 epochs

tion, my intention was to implement them in a **Keras** callback function, unfortunately I did not manage to access a **Pandas** DataFrame from the callback, for this reason I used the accuracy as metrics of the network and I tried to implement the custom metrics as separate functions to run on the testing dataset. Despite my efforts this measure does not perform as the accuracy, in fact I can see a value of 0.026378 on the first measure and 0.022458 on the second measure. I did not have the time to explore this issue, for this reason I rely on the simple accuracy.

The main code for the training can be found in the Assignment_3.py file.

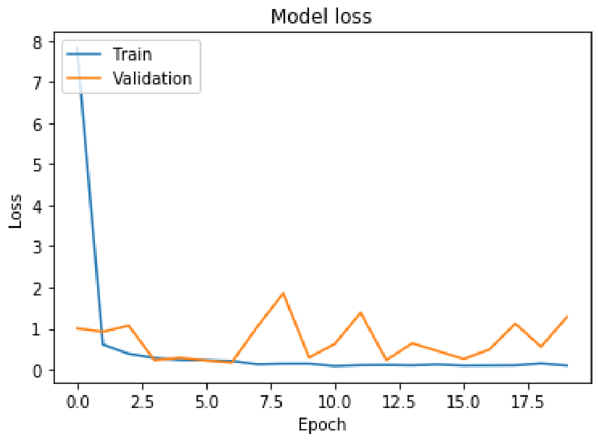


Figure 4: Loss 20 sample classes over 20 epochs

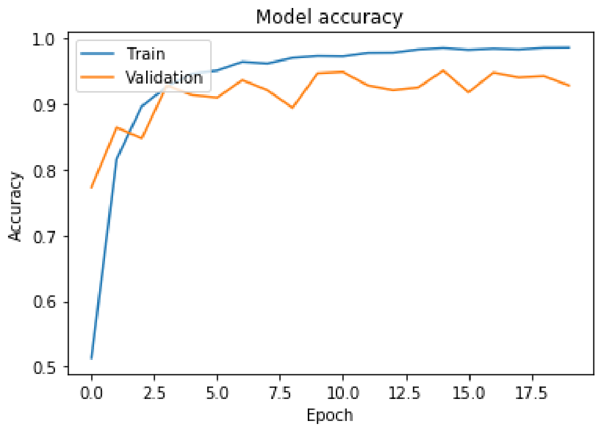


Figure 5: Accuracy 30 sample classes over 20 epochs

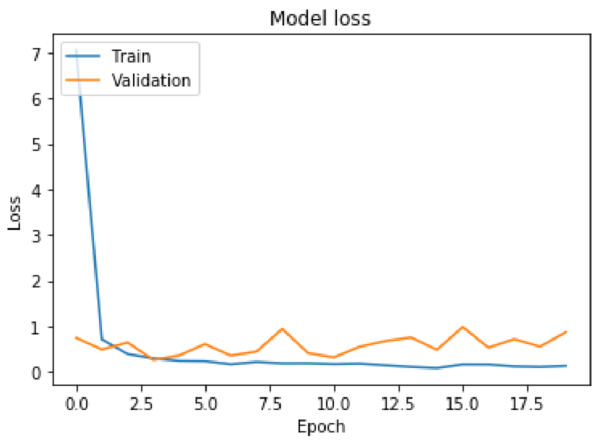
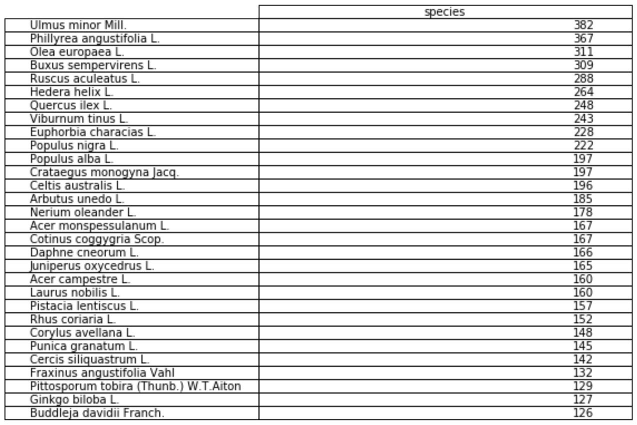


Figure 6: Loss 30 sample classes over 20 epochs



	species
Ulmus minor Mill.	382
Phillyrea angustifolia L.	367
Olea europaea L.	311
Buxus sempervirens L.	309
Ruscus aculeatus L.	288
Hedera helix L.	264
Quercus ilex L.	248
Viburnum tinus L.	243
Euphorbia characias L.	228
Populus nigra L.	222
Populus alba L.	197
Crataegus monogyna Jacq.	197
Celtis australis L.	196
Arbutus unedo L.	185
Nerium oleander L.	178
Acer monspessulanum L.	167
Cotinus coggygria Scop.	167
Daphne cneorum L.	166
Juniperus oxycedrus L.	165
Acer campestre L.	160
Laurus nobilis L.	160
Pistacia lentiscus L.	157
Rhus coriaria L.	152
Corylus avellana L.	148
Punica granatum L.	145
Cercis siliquastrum L.	142
Fraxinus angustifolia Vahl	132
Pittosporum tobira (Thunb.) W.T.Aiton	129
Ginkgo biloba L.	127
Buddleja davidii Franch.	126

Figure 7: The first 30 sample classes

References

[1] Hartley and Zisserman. *Multiple View Geometry in Computer Vision*. 2004.

[2] Tiago S. Nazare Tu Bui John Collomosse Moacir A. Ponti, Leonardo S. F. Ribeiro. *Everything you wanted to know about Deep Learning for Computer Vision but were afraid to ask*. 2017.