



REVUE ALGERIENNE DES SCIENCES

Section -A-

Sciences de la Nature et de la Vie Sciences techniques

<http://univ-eltarf.dz/fr/>



Object multi-detection and multi-tracking via machine learning and deep learning

LabeniMerouane^{a,*}, BoufekarChaouki^b, TaffarMokhtar^a

^aDepartment of Computer Science, Mohammed Seddik BenYahia University, Jijel, Algeria.

^bDepartment of Computer Science, Algiers 1 University, Algiers, Algeria.

Information	Abstract
Keywords: Image/Video Processing Machine Learning Deep Learning *Correspondence labmer@gmail.com	Objects detection and tracking in video frames constitute a vital field of research that provide appropriate information about objects and their trajectories. An interesting technique to achieve these goals is using digital image/video processing methods with machine learning and deep learning methods. This paper presents an object multi-detection and multi-tracking approach based on a combination of machine learning methods, using KMeans clustering and silhouette coefficient measure that are used for object multi-detection with a deeper convolutional neural network method which represents a ConvNet architecture used for object classification, and a proposed algorithm using a mathematical concept for object multi-tracking. This work demonstrates that the mission of object multi-detection and multi-tracking can be achieved via standard image/video processing and machine learning (consequently deep learning) methods. Therefore, and for both detection and tracking goals, we will show that optical flow analysis, data analysis and ConvNets architectures are well suited towards these goals.

1. Introduction

Image/video processing is the field of handling and analyzing images. There are several hidden information in an image that we unconsciously process. The goal of image/video processing is to use the different properties of an image/frame such as color, correlations between different pixels, pixels motion, object number and their placements.

Convolutional Neural Network (also called ConvNet) is one kind of feed-forward neural networks. It is an efficient recognition process which is widely used in pattern recognition and image processing. It has many features such as simple structure, less training parameters and adaptability. The pre-trained VGG19 [1] ConvNet used in this paper for image identification is a deeper convolutional neural network that is trained on the ImageNet database [2].

Several works, using image/video processing and deep learning methods for object identification, are mentioned these last years, especially in recent deep ConvNets (CNNs) architectures that got an important improvements, these last allow them to outperform traditional computer vision techniques in image

detection and classification tasks across benchmark datasets [3].

A recent ConvNet model that is called R-CNNs (for Regional Convolutional Neural Networks) combine these two tasks; candidate object locations are determined and automatically classified [4]. However, in its original form, R-CNNs are computationally expensive to train and evaluate. An important idea is used to fix this challenge that consists on the introduction of sharing convolutions across proposals in Fast R-CNN and SPPnet [5]. In [6] Ren et al. propose the addition of a Region Proposal Network (RPN), which shares convolutional features with the detection network leading to Faster-RCNN. Furthermore, Region-based Fully Convolutional Networks (R-FCNs) [7].

In [8] an approach was proposed for foreground (i.e. object) extraction with minimal user interaction. Initially user draws a rectangle around the object region. Then GrabCut algorithm segments it iteratively to get the best result. The main difficulty in this approach is that, in some cases the segmentation won't be fine, like, it may have marked some foreground region as background and vice versa. In that case, user needs to do fine touchups with whitestrokes (denoting foreground) and black strokes (denoting background).

In [9] the principal goal is to detect and locate Friesian cattle in images obtained from a top-down or aerial standpoint. The deep network used to address this problem is the R-CNN adaptation of the VGG CNN M 1024 network published as part of several other network architecture proposals [10]. In another approach called YOLO described in [11] the authors frame object detection as a regression problem to spatially separated objects (i.e. Bounding boxes) and associated class probabilities, this model imposes strong spatial constraints on bounding box predictions since each grid cell only predicts two boxes and can only have one class. This spatial constraint limits the number of nearby objects that the model can predict. YOLO based algorithm with Gaussian Mixture Model by using the concepts of deep learning has given good accuracy for feature extraction and classification [12].

In [13] an approach based on SSD [14] and MobileNets [15] algorithms for object detection and tracking. With a help of RoIs (region of interest) the object is detected from given class of image. Different methods are used: Frame differencing, Optical flow, Background subtraction, this last is constrained by using a static camera.

Comparing the model proposed in this paper to other detection systems like [9] [11-13] we can highlight the main difference adopted by our approach that consists on the use of the VGG19 [1] model power and KMeans clustering method with the math rigorous concepts to allow specific objects selection ensuring a best object multi-detection and multi-tracking. However our approach gives alternate and weaker results in some cases that disappear in other detection systems[9][12,13] : In the case of small object detections (i.e. VGG19[1] model used for identification deals with large-scale images) and in the case of several objects within nearby frames that have mixed positions.

2. Tools and methods

2.1. Tools

2.1.1. Python OpenCV

OpenCV is a C++ library with modules that cover many areas of computer vision (image and video processing). Besides C++ (and C) there is growing support for Python as a simpler scripting language through a Python interface on top of the C++ code base [16]. This gives us two advantages: first, the code is as fast as original C/C++ code (since it is the C++ code working in background) and second, it is very easy to code in Python. So OpenCV-Python is an appropriate tool for fast prototyping of computer vision problems [17].

2.1.2. Opticalflow

Optical flow (sometimes called optic flow) is the image motion of objects as the objects, scene or camera moves between two consecutive images. It is a 2D vector field of within-image translation. It is a classic and well-studied field in computer vision with many successful applications in for example video compression, motion estimation, object tracking and image segmentation [16].

Consider a pixel $I(x, y, t)$ in first frame (Check a new dimension, time, is added here). It moves by distance (dx, dy) in next frame taken after dt time. So since those pixels are the same and intensity does not change, we can say [17]:

$$I(x, y, t) = I(x + dx, y + dy, t + dt) \quad (1)$$

OpenCV contains several optical flow implementations, we then use *calcOpticalFlow – Farneback* method based on [18]. That is considered one of the best methods for obtaining dense flow fields [16].

2.1.3. Slicing operation

Slicing is a Numpy (i.e. Numeric python library) operation that allows us to extract a sub-object from an object [19].

An image $I = \{ \{1, \dots, n_1\} * \{1, \dots, n_2\} \rightarrow W \subseteq \mathbb{R} \}$ can be represented by a 2d-array of size $n_1 \times n_2$. Often, W is the set $\{0, \dots, 255\}$ representing a 08 bits channel (gray scale), then a color image can be represented by a 3d-array of size $n_1 \times n_2 \times 3$ (i.e. *width* $\times$ *height* $\times$ *channels*) assuming three color channels (e.g. RGB) [20]. In these cases a slicing operation can have –within Numpy library– one of two following forms:

- *Sub-image* = *image*[*a:b, c:d*] in gray scale
- *Sub-image* = *image*[*a:b, c:d, :*] in color image

Where $a \geq 0$, $b \leq n_1$, $c \geq 0$ and $d \leq n_2$.

2.1.4. Videoprocessing and object tracking

Videos are a sequence of images and hence dealing with videos is similar to how we deal with images with a few exceptions of course, tracking objects in a video is a field of video processing [21], given the initial set of points (region of interest), a tracker tries to calculate the motion of these points by looking at the direction of change in the next frame (e.g. the KCF [22] tracker).

2.1.5. Machine learning and deep learning

Modern artificial intelligence systems and machine learning algorithms have revolutionized approaches to scientific and technological challenges in a variety of fields. We can observe remarkable improvements in the quality of state-of-the-art computer vision, natural

language processing, speech recognition and other techniques [23]. Deep learning (DL) is a particular subset of Machine Learning methodologies using artificial neural networks (ANN) slightly inspired by the structure of neurons located in the human brain. Informally, the word deep refers to the presence of many layers in the artificial neural network [24].

Convolutional neural network (CNN) is a technique of deep learning that consists on different types of layers (principal components of a CNN), based on these layers complex architectures are used for classification tasks which are built by stacking multiples-layers [22], each layer learns to identify the features that are necessary to do the final classification [25].

2.1.6. Scikit-learn and Keras libraries

Since its release in 2007, scikit-learn has become one of the most popular open source machine learning libraries for Python. It provides algorithms for machine learning tasks including classification, regression, dimensionality reduction, and clustering. Scikit-learn is built on the popular Python libraries NumPy and matplotlib. NumPy extends Python to support efficient operations on large arrays and multidimensional matrices. matplotlib provides visualization tools [26].

Keras (i.e. [tf.keras](https://keras.io)) is actually the TensorFlow's [27] high-level API for building and training deep learning models. It performs modular-building models by stacking layers and connecting computational graphs. Keras is useful for fast prototyping (i.e. Ignoring the details of implementing different algorithms or writing optimization procedures) that supports convolution and recurrent layers and combination of both, it runs seamlessly on CPUs and GPUs with almost any architecture can be designed using this library [28]. The VGG19 [1] model used in this paper is one of the Keras pretrained models.

2.1.7. KMeans clustering and silhouette coefficient measure

Clustering, or cluster analysis, is the task of grouping observations such that members of the same group, or cluster, are more similar to each other by a given metric than they are to the members of the other clusters [26]. The K-Means algorithm is a clustering method that is popular because of its speed and scalability. The titular k is a hyperparameter that specifies the number of clusters that should be created. K-Means automatically assigns observations to clusters but cannot determine the appropriate number of clusters (i.e. k) [26].

For this we will use a performance measure for clustering called the silhouette coefficient. The silhouette coefficient is a measure of the compactness and separation of

the clusters. It increases as the quality of the clusters increase, it is large for compact clusters that are far from each other and small for large, overlapping clusters. [26].

2.1.8. Barycenter

Giving a set of points $A_1, A_2, \dots, A_n \in E$ and some real numbers $\lambda_1, \lambda_2, \dots, \lambda_n$ of not null sum, it exists a single point $G \in E$ verifying [29]:

$$G = \sum_{i=1}^n \lambda_i A_i \quad (2)$$

Where $\sum_{i=1}^n \lambda_i = 1$.

This point is called Barycenter of $A_1, A_2, \dots, A_n$ points affected by $\lambda_1, \lambda_2, \dots, \lambda_n$ coefficients.

2.2. Methods

The methodology proposed in this paper uses a combination of a proposed approach-basing on optical flow handling and Kmeans clustering- for "object multi-detection and multi-tracking" with a "pretrained VGG19 [1] ConvNet" for objects classification. The pre-trained VGG19 [1] ConvNet is used to classify each detected object then we pass its classification back to use as object label. A tracking algorithm takes place and waits for all objects that have the classification confidences surpass an acceptance threshold for starting object multi-tracking, the main idea in this last consists on tracking the barycenter of l fastest pixels within each object (i.e. bounding box). The architecture of the proposed method is presented in Figure 1.

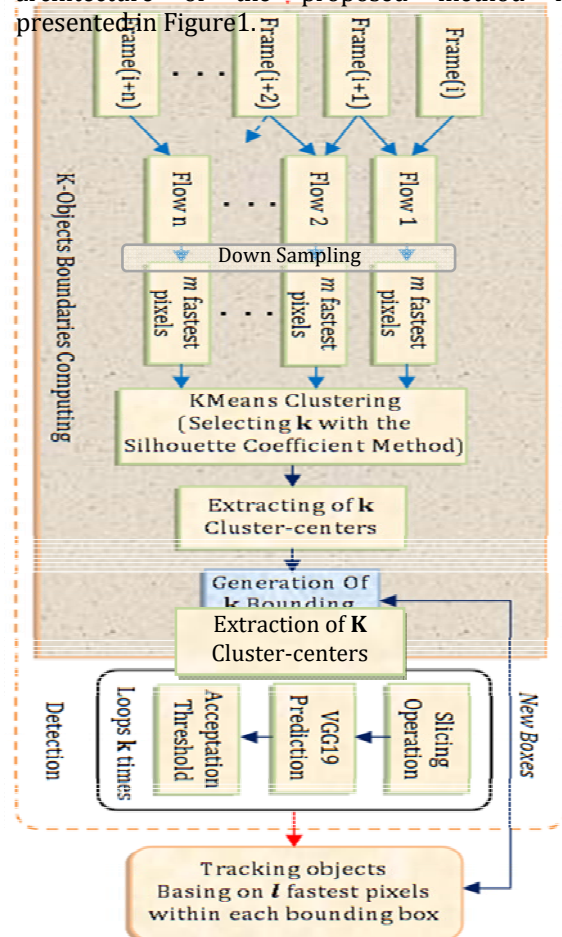


Fig. 1: The architecture of the technique

Based on Figure 1, the technique includes the following steps: Optical flow handling, KMeans clustering, bounding boxes generating, sub-images extracting using slicing, VGG19[1] classifications and finally object multi-tracking.

2.2.1. Optical flow handling and KMeans clustering

After reading of each frame in its original size (1st level) it is down sampled twice (2nd and 3rd levels) using the OpenCV function `pyrDown()` which if no new size is given, creates a new image half the size of the original (in 3rd level a pixel represents 4x4 pixels in 1st level), we use downsampling for increasing detection process efficiency and speed (i.e., increasing opportunity for slow objects to be captured), then we use the `calcOpticalFlowFarneback` method based on [18] for computing optical flow generated between frames (see Fig. 2), this method is used to detect objects which are in motion across frames using a camera or video records, for that we take $(n + 1)$ frames and compute the optical flow between each doublet of frames, then we extract m fastest pixels that have the fastest motions (represented by either dx in X-axis or dy in Y-axis) for each generated flow of n flows. Thus we obtain $n \times m$ fastest pixels.

We then pass these $n \times m$ points (i.e. pixels) as first parameter to the `centroids()` function described in Figure 3, and within this function we call the KMeans method for each value of a test vector = [2, 3, 4, 5, 8] (possible values of k) and we calculate the associated silhouette coefficients, then we extract the maximum coefficient (that explains the right value of k) with the associated clustering scheme (i.e. number of clusters, the pixels' membership and their centers).

```
def coordinates(n,m,w,h):
    pixels=[];coord=[];pyrdown=4
    _,im = cap.read()
    imh2=cv2.pyrDown(im);imh4=cv2.pyrDown(imh2)
    old_gray = cv2.cvtColor(imh4,cv2.COLOR_BGR2GRAY)
    for l in range(n):
        _,im = cap.read()
        imh2=cv2.pyrDown(im);imh4=cv2.pyrDown(imh2)
        gray = cv2.cvtColor(imh4,cv2.COLOR_BGR2GRAY)
        # compute flow
        flow = cv2.calcOpticalFlowFarneback(old_gray,gray,\
                                           None,0.5,3,15,3,5,1.2,0)
        print('w x h =',w,'x',h,'Total :',w*h//pyrdown**2)
        old_gray = gray
        for q in range(m):
            fastestmotion=max(abs(flow.ravel()))
            for i in range(w//pyrdown):
                for j in range(h//pyrdown):
                    if (abs(flow[j,i,0]) == fastestmotion or abs(flow\
                                                                    [j,i,1]) == fastestmotion):
                        print('x:',i*pyrdown,'y:',j*pyrdown,'dxdy:',flow[j,i])
                        pixels.append((i*pyrdown,j*pyrdown))
                        flow[j,i,0]=0;flow[j,i,1]=0
            maxslh,_tuple=centroids(pixels,[2,3,4,5,8],n)
            print('***Silhouettel :',maxslh,_tuple);print(_tuple[2])
            coord=compute(_tuple)
            print('$$$Coord ',coord)
    return coord,im
```

Fig. 2: Code for optical flow computing and fastest pixel extracting.

In this step we fix detection at four objects, this is explained by the fact that the VGG19[1] model deals with large-scale images and not small ones, then for all values of k that surpass this barrier the detection process will be repeated but this time with the clusters' centers themselves.

```
def centroids(points,tests,n):
    silhouette=dict()
    for t in tests:
        kmeans_model = KMeans(n_clusters=t).fit(points)
        silhouette[metrics.silhouette_score(points,kmeans_ \
        model.labels_, metric='euclidean')]= (t,kmeans_model. \
        labels_,kmeans_model.cluster_centers_)
    return max(silhouette), silhouette[max(silhouette)]
```

Fig. 3: Code for centroids function

2.2.2. Bounding boxes generating and images slicing

In this step we pass the k clusters' centers returned par the `KMeans()` method to the `compute()` function (see Fig. 4) responsible for the bounding boxes definition, for that and for each cluster's center this function calls another function called `boundary()`, using a fixed offset, this last computes the associated bounding box like illustrated in Figure 5.

```
def compute(_tuple):
    computed=[];centers=[]
    k=_tuple[0];boolean=False
    for j in range(k):
        clustcenter=_tuple[2][j]
        if k<=4:
            computed.append(boundary(clustcenter,offset=180))
            print('computed:',computed)
        else:
            centers.append(clustcenter)
            boolean=True
    if boolean:
        maxslh,_tuple=centroids(centers,[2,3,4],0)
        print('***Silhouette2 :',maxslh,_tuple)
        return compute(_tuple)
    return computed
```

Fig. 4: Code for bounding boxes computing

Figure 5 shows detection process basing on clusters' centers and offset value, note that the top of each bounding box is weighted against other sides this is because (by intuition and for the most of objects) the bottom side moves more than other sides, in this case the cluster's center becomes the bounding box' pseudo-center, we keep this idea unchanged during tracking. The values of x_1, y_1, x_2, y_2 are computed by the *boundary()* function illustrated in Figure 6.

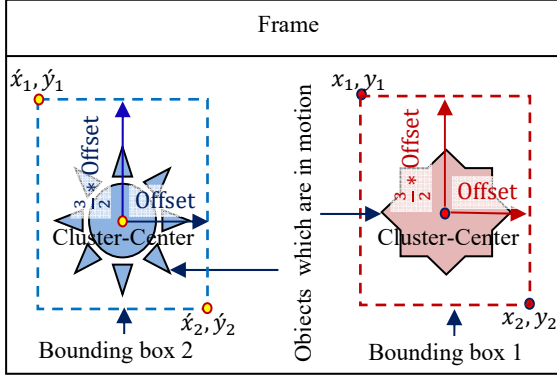


Fig. 5: Clusters' centers and associated bounding boxes

```
def boundary(clustercenter, offset):
    x1=int(clustercenter[0]-offset)
    y1=int(clustercenter[1]-offset*3/2)
    x2=int(clustercenter[0]+offset)
    y2=int(clustercenter[1]+offset)
    if x1<0:x1=0
    if y1<0:y1=0
    if x2>w:x2=w-1
    if y2>h:y2=h-1
    return x1,y1,x2,y2,clustercenter
```

Fig. 6: Code for *boundary* function.

For determining the sub_images (i.e. detected objects) basing on these bounding boxes we use slicing operation explained above.

In our case we use 3d-array images (i.e. video color frames) then a slicing operation through these images we gives a 3d-array sub-images (see Fig. 7).

```
while True:
    k=0;detect=[];_,im = cap.read();old_im=im
    coordvec,im=coordinates(n,m,w,h)
    for i in range(len(coordvec)):
        x1,y1,x2,y2,cent=coordvec[i][0],coordvec[i][1],coordvec[i][2]\
            ,coordvec[i][3],coordvec[i][4]
        box=old_im[y1:y2,x1:x2,:]
        cv2.imwrite('img'+str(k)+' .png',box)
        pr=vvg19('img'+str(k)+' .png')
        if pr[0] > threshold :
            detect.append((pr[0],pr[1],(x1+50,y1+50,x2-50,y2-50,cent)))
        k+=1
    print('Detected Objects',detect)
    if len(detect)!=0:
        tracker(detect,im,n,w,h);Break=True
    break
```

Fig. 7: Code for sub_images extraction using slicing

2.2.3. The Vgg19 function

We define the *vvg19* function (see Fig. 8) within a python module called *vvg.py* in the python site-packages directory, this function uses

the pretrained VGG19 [1] model for predicting sub_images (i.e. detected objects).

This function returns the prediction that has the maximum probability among probabilities returned by the pretrained VGG19 [1] ConvNet, these probabilities explain the membership percentage of the detected objects at each class in the output layer of the VGG19 [1] model.

```
from tensorflow.keras.applications.vgg19\
    import VGG19, decode_predictions
....
from numpy import *
def vvg19(detectedobject) :
    model = VGG19()
    ....
    y_pred = model.predict(image)
    labels = decode_predictions(y_pred)
    predictions=dict()
    for i in range(5):
        predictions[labels[0][i][2]]=(labels[0]\
            [i][1],labels[0][i][0])
    idd=max(predictions.keys())
    return round(idd,3),predictions[idd][0]
```

Fig. 8: Code for *vvg19* function

2.2.4. Object multi-tracking

In the tracking process our model tracks objects by capturing l fastest pixels (represented by tracking_pixels variable, see Fig. 9) in each associated bounding box (i.e. the object container) and computing their barycenter using equation (2) with their motions as coefficients (i.e. $\lambda_i = \text{fastestmotion}_i / \text{coefficients}$ where $\text{coefficients} = \sum_{i=1}^n \text{fastestmotion}_i$). The computed barycenter is used for the next bounding box definition. The system starts tracking by calling the *tracker()* function which has five parameters (see Fig. 7): list of detected objects (with their classifications and bounding boxes), actual frame, optical flows' number, frame width and height.

In the tracking process and for increasing model performance, especially in the case of objects interference during tracking, we reduce the tracking area to a part of each detected object. In this case the bounding boxes' sizes are computed according to the pixels fastest motion within each bounding box.

```
fast=[];tracking_pixels=4
for l in range(tracking_pixels):
    fastestpixel=max(abs(fl.ravel()))
    for i in range(detect[k][2][0],detect[k][2][2]):
        for j in range(detect[k][2][1],detect[k][2][3]):
            if (abs(flow[j,i,0]) == fastestpixel or abs(flow[j,i,1]) ==\
                fastestpixel):
                print('x=',i,'y=',j,'dx dy=',flow[j,i]);flow[j,i]=0
                fast.append((i,j,fastestpixel))
    bary_,fastmotion=bary(fast);print('Motion:',fastmotion)
```

Fig. 9: Code for fastest pixels extraction

3. Results and discussion

For illustrating the utility of our model we use two scenarios: *detection* and *tracking*, in the first scenario we execute the model with help of videos records (dynamic camera) and with a

notebook camera (static camera), based on fastest pixels' number m , and the optical flows' number n . The acceptance threshold is fixed at 0.2 (i.e. 20% of confidence). In the second scenario we ignore the acceptance threshold and we concentrate on tracking operation.

For the first scenario we plot, with help of matplotlib Python library, the clusters' forms returned by the *KMeans* method with the associated silhouette coefficients for each value of the test vector (possible values of k).

3.1. Scenario1: Detection

3.1.1. Using video records (dynamic camera)

Figure 10 shows detection of a single object basing on 100 fastest pixels and one flow, note that the camera speed should be lower than object speed, in case the camera has a nearby speed with the object we must increase the fastest pixels number and flows number.

In the *KMeans* algorithm the minimum number of clusters k should be upper or equal to two clusters, then in case of single object detection the object will be detected twice, this last is solved by the system itself during tracking process by integrating detections that have the same classification, like illustrated in Figure 11.

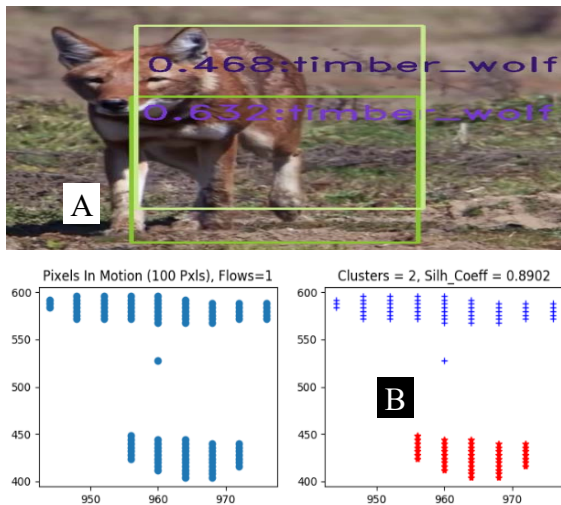


Fig. 10: A: Detection of timber_wolf (63.2% and 46.8%), B: the associated KMeans clustering with the maximum silhouette coefficient.



Fig. 11: One object detection with labels unification.

In this last step the system compares the detections and chooses the detection that has the great percentage of confidence.

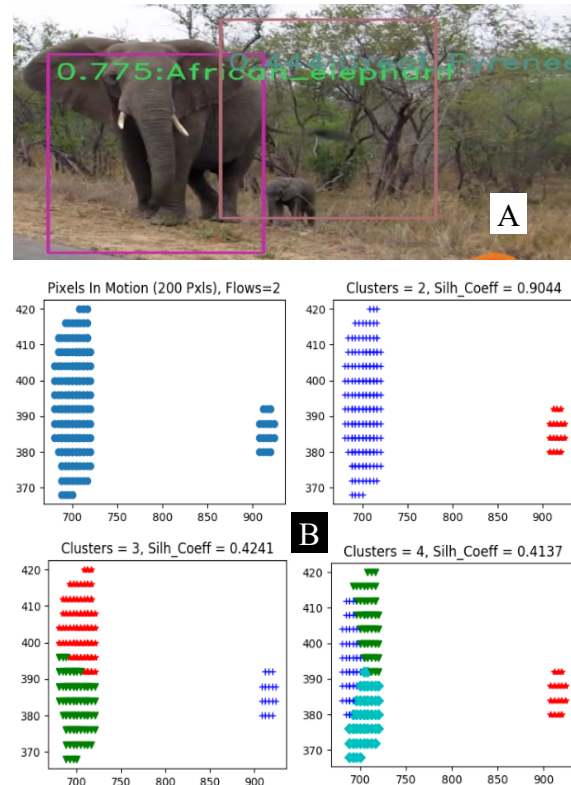


Fig. 12: A: Detection of African_elephant (77.5%) and Great_Pyrness (44.4%), B: The associated KMeans clustering with the silhouette coefficients until 04 clusters.

As we can see in Figure 12, the system detects two clusters representing the elephant (successful identification) and its tail (mismatch identification) basing on 200pixels (100 pixels per flow).

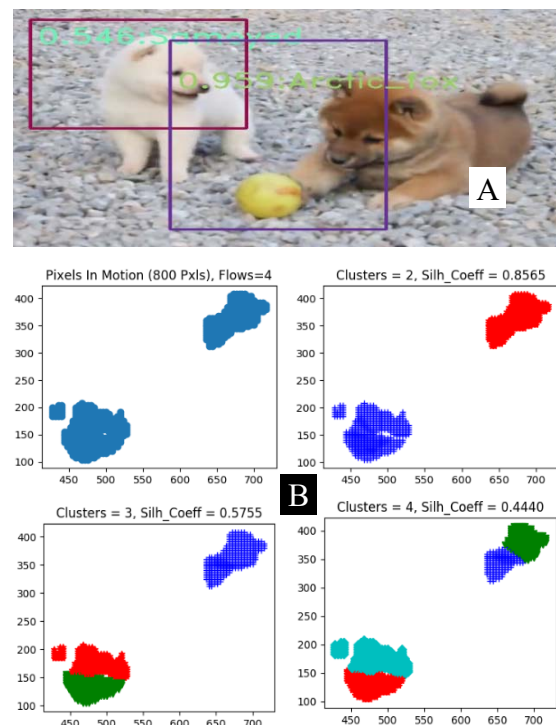


Fig. 13: A: Detection of Arctic_fox (95.9%) and Samoyed (54.6%), B: The associated KMeans clustering with the silhouette coefficients until 04 clusters.

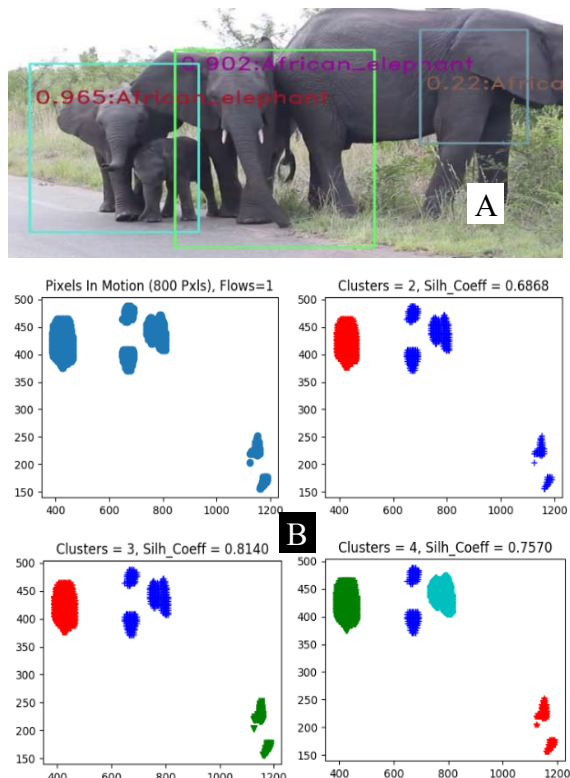


Fig. 14: A: Detection of three African_elephants(96.5%, 90.2% and 22%), B: The associated KMeans clustering with the silhouette coefficients until 04 clusters.

3.1.2. Using notebook camera

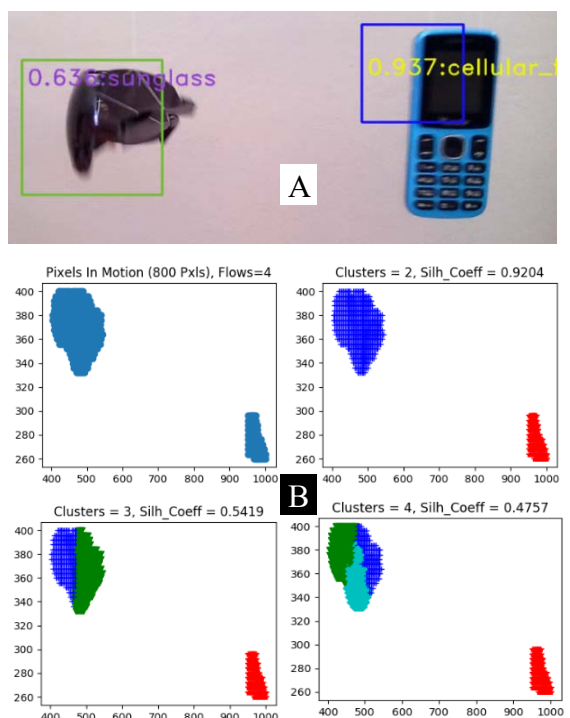


Fig. 15: A: Detection of Sunglass (63.6%) and Cellartelephone (93.7%), B: The associated KMeans

clustering with the silhouette coefficients until 04 clusters.

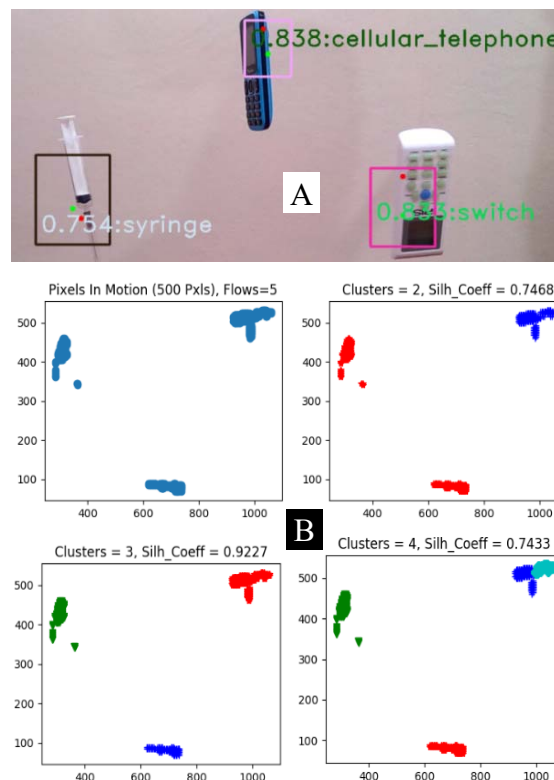
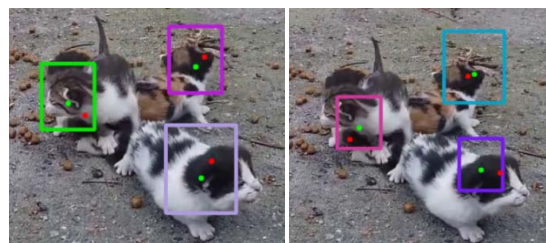


Fig. 16: A: Detection of Switch (83.3%), Cellartelephone (83.8%) and Syringe (75.4%), B: The associated KMeans clustering with the silhouette coefficients until 04 clusters.

During detection and for both dynamic camera and static camera the technique has given well results.

3.2. Scenario 2: Tracking

In this scenario we ignore object identification and we highlight the tracking process. The object multi-tracking process is achieved by computing the barycenter (note that this technique retards the bounding boxes union) of the fastest pixels in each bounding box (during tracking we also reduce the bounding boxes' sizes for avoiding their interference thus their union). Figure 17 illustrates this process.



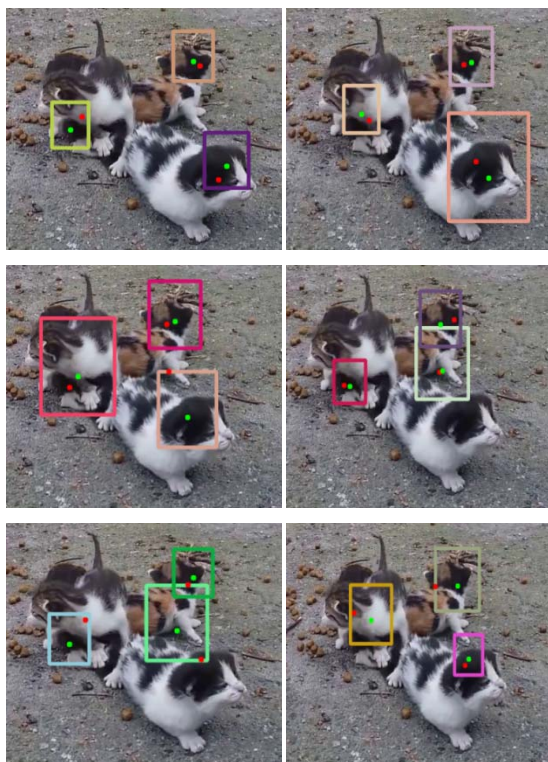


Fig. 17: Object multi-tracking.

The system tracks a part -represented by the colored rectangle- of each object, the green points represent the actual boxes' pseudo-centers (see Fig.5) and the red ones represent the l fastest pixels' barycenter within each actual bounding box (i.e. the next bounding boxes' pseudo-centers).

The object multi-tracking process gives acceptable results except in the case of forte object interference where the bounding boxes that contain the same l fastest pixels will be integrated.

4. Conclusion

In this paper an approach for object multi-detection and multi-tracking has been proposed, an implementation of the proposed approach has been performed using python programming language basing on machine learning and deep learning libraries.

Object detection and tracking are basic data for security and inspection process which includes surveillance tasks. However, processing within limited and real time video frames is challenging task for prevention, surveillance and scientific purposes, likewise the real time analysis of the technique can yield great results by enabling security in all supervision domains.

In the future and continuing improvement of the work we will concentrate on the amelioration of the system interaction in the case

of an interference's high level basing on image-video processing algorithms.

References:

- [1] Simonyan, K., Zisserman, A. (2014) Very deep convolutional networks for large-scale image recognition. CoRR, vol. abs/1409.1556.
- [2] Deng, J., Dong, W., Socher, R., Li, L.-J., Li K., and Fei-Fei. L. (2009) Imagenet: A large-scale hierarchical image database. In Computer Vision and Pattern Recognition. CVPR 2009. IEEE Conference on, pages 248–255.
- [3] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012) Imagenet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems 25, pages 1097–1105. Curran Associates, Inc.
- [4] Girshick, R. B., Donahue, J., Darrell, T. and Malik, J. (2013) Rich feature hierarchies for accurate object detection and semantic segmentation. CoRR, abs/1311.2524.
- [5] He, K., Zhang, X., Ren, S. and Sun, J. (2014) Spatial pyramid pooling in deep convolutional networks for visual recognition. ArXiv preprint, arXiv:1406.4729.
- [6] Ren, S., He, K., Girshick, R. and Sun, J. (2015) Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. ArXiv 2015, pages 1–10.
- [7] Dai, J., Li, Y., He, K. and Sun, J. (2016) R-FCN: object detection via region-based fully convolutional networks. CoRR, abs/1605.06409.
- [8] Rother, C., Kolmogorov, V., and Blake, A. (2004) "GrabCut": interactive foreground extraction using iterated graph cuts. Microsoft Research Cambridge, UK. ACM Transactions on Graphics (SIGGRAPH).23:309–314.
- [9] Andrew, W., Greatwood, C., and Burghardt, T. (2017) Visual localisation and individual identification of holstein friesian cattle via deep learning. In IEEE International Conference of Computer Vision Workshop ICCVW, pp. 2850–2859.
- [10] Chatfield, K., Simonyan, K., Vedaldi, A. and Zisserman, A. (2014) Return of the devil in the details: Delving deep into convolutional nets. In British Machine Vision Conference.
- [11] Redmon, J., Divvala, S., Girshick, R., Farhadi, A. (2016) You only look once: unified, real-time object detection. IEEE Conference on Computer Vision and Pattern Recognition.
- [12] Jana, A. P., Biswas, A., Mohana, M. (2018) YOLO based detection and classification of objects in video records. IEEE International Conference On Recent Trends In Electronics Information Communication Technology (RTEICT), India.
- [13] Chandan, G., Jain, A., Jain, H., Mohana, M. (2018) Real time object detection and tracking using deep learning and OpenCV. Proceedings of the International Conference on Inventive Research in Computing Applications, ICIRCA.
- [14] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C. Y., Berg, A. C. (2016) SSD: Single shot multibox detector. ArXiv:1512.02325v5 [cs.CV].

- [15] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H. (2017) MobileNets: Efficient convolutional neural networks for mobile vision applications. ArXiv:1704.04861v1 [cs.CV].
- [16] Solem, J. E. (2012) Programming computer vision with Python. Creative commons. United states license, USA.
- [17] Mordvintsev, A., Abid, K., (2017) OpenCV-Python. Tutorials documentation, Release 1.
- [18] Farnebäck. G. (2003) Two-frame motion estimation based on polynomial expansion. In Proceedings of the 13th Scandinavian Conference on Image Analysis, pages 363–370.
- [19] Seppke, B. (2013) Image processing with Python: An introduction to the use of Python, NumPy, SciPy and matplotlib for image processing tasks. Hamburg University, Germany.
- [20] Stutz, D. (2014) Understanding convolutional neural networks. Seminar report.
- [21] Kapur, S. (2017) Computer vision with Python 3. Packt Publishing Ltd. Prod. Ref: 1210817. Birmingham B3 2PB, UK.
- [22] Henriques, J. F., Caseiro, R., Martins, P. and Batista, J. (2015) High-speed tracking with kernelized correlation filters. IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI). Vol. 37. No. 3, pp. 583–596.
- [23] Goldsborough, P. (2016) A tour of tensorflow. Pro-seminar Data Mining. ArXiv:1610.01178v1 [cs.LG].
- [24] Golnari, A. (2019) Introduction to deep neural networks with Keras. ResearchGate.
- [25] Gulli, A., Pal, S. (2017) Deep learning with Keras-Implement neural networks with Keras on Theano and Tensorflow. Packt Publishing Ltd. Birmingham, Mumbai.
- [26] Hackeling, G. (2014) Mastering machine learning with Scikit-learn. Packt Publishing Ltd, Prod. Ref: 1221014, Birmingham B3 2PB, UK.
- [27] The TensorFlow website. [Online]. available : <https://www.tensorflow.org/>
- [28] Sathrey, K. (2018) Tutorial on Keras. CAP 6412 - Advanced Computer Vision.
- [29] Deschamps, C., Warusfel, A., Moulin, F., Ruaud, J. F., Miquel, A., Sifre, J. C. (2003) Mathematics All-In-One. French edition: Mathématiques Tout-En-Un. Dunod. Paris. ISBN 978 2 10 00 7944 5, 2^e edition.