Giulia Rizzoli 1241721

Laboratory 5 (Final Project)

Execution order:

- Run CIFAR10Train.ipynb (Train CNN)
- Run Lab5.cpp (Extract Frames)
- Run FinalClassification.ipynb (Classify pictures' frames)

1- Frame extraction:

Task: Develop a computer vision algorithm detecting the photo frames in C++ with the OpenCV library.

Main strategy to detect photo frames:

1 – Filtering: smoothing the image to remove noise. Blur() function is used to have less parameters to tune. Kernel size is set to (3,3) in order to not alter too much the images content. In other words to preserve the edges. Note: if filtering too much the results are going to be even more approximated, but sometimes it is strictly necessary to clean out un-necessary structures (as for hard3 image).
2 – Grey scale conversion: processing in order to apply thresholding or edge detector.
3 –Edge detection: gather the edges of the image with canny edge detector. Otsu's thresholding is used after canny since it seems it corrects a bit more the quantization of contours.
4 – Find rectangle structures: use a combination of the function findContour (similar to Hough Transform, but more precise for this purpose) and approxPoly, which together gather the contours and given the contour points, compute an approximation of the polygons.
Then, since the frames are all rectangles or squares, only 4 vertex polygons are selected. A bound on minimum and maximum area is imposed: it picks only relevant squares, which means that the algorithm discards frames that are too small (we are almost sure that they are not a frame) and also big frames, e.g. the whole image can't be considered a frame.

The code work as follow:
It receives in input a path (the path folder needs to contain ONLY the images for this specific task otherwise all the other images will be opened).
As an alternative, if no input path is selected, the program will search for the six images in the Project folder (the set of images' names is given).

- For each image (one at the time) the program opens a window which shows the original image and the same image with canny+thresholding applied. The window contains also two trackbars that can be used to tune two of the main parameters of canny edge detector: the first is the minimum threshold parameter, the other is the ratio, defined in such a way that threshold_h = ratio*threshold_l. If parameters are tuned correctly, one or more coloured rectangles will appear on the original image, in the position where the frames are supposed to be.
- RIGHT click on the window to select the current parameters as the desired ones. By doing so the frames will be extracted. Then the window will close.
- The next window for the next image pops up.

This procedure iterates until the parameters for the last image are selected.
In the end the frames are resized and saved in the folder "../Frames" inside the Project folder.

Notes:

- It can be noticed that in some cases, as in Easy1 and Easy3 images, the picture frames can be simply extracted by using Otsu thresholding. In both images, the border shape is very defined and there is a significant change in shade between the picture and the wall.

Figure: Otsu Thresholding vs Canny Edge Detection on Easy1 image.

In other images Otsu doesn't work directly since it is not a very robust approach to illumination effects and change of shades (see for example Easy2 image).

- Some of frames are thick-edged therefore the frame could be captured twice (one rectangle for the inner frame border and another for the outer one) or even more as in the case of Horse picture.

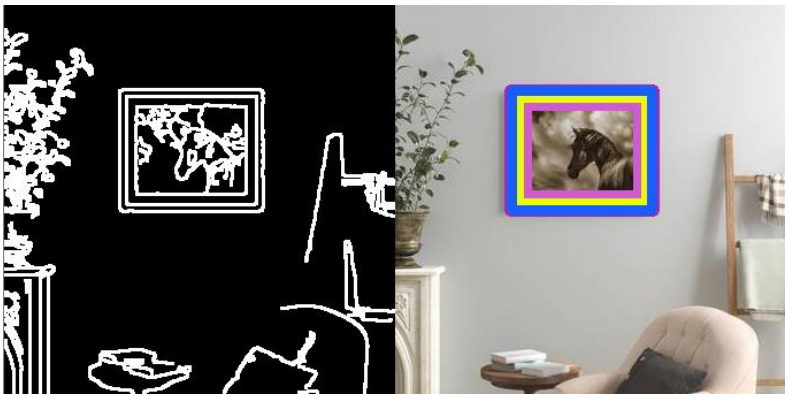


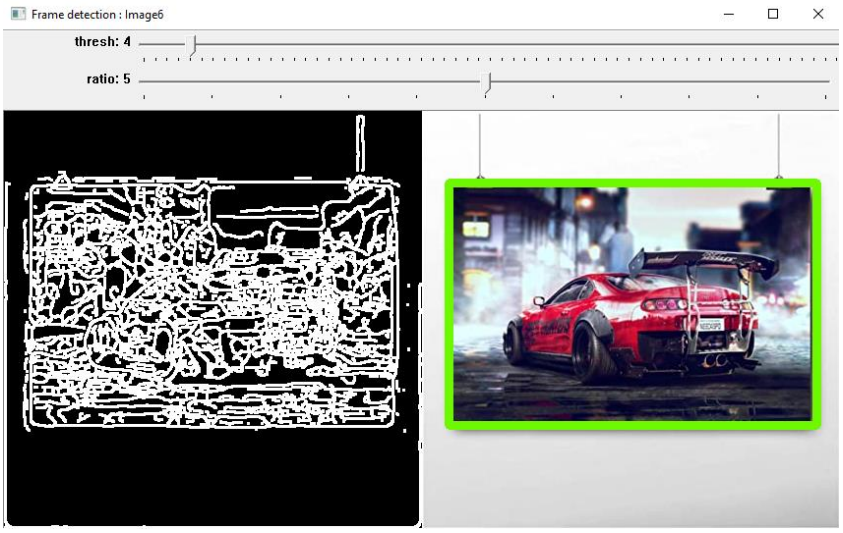
Figure: Multiple frames captured on Hard1 image.

- In some cases, other rectangle-like structures are captured in the image.

Figure: False frames detection on Hard4 and Easy4 images.

In those cases, the lower threshold needs to be increased to clean out weak edges or non-interesting structures. The code is implemented in a way that increasing the lower threshold make greater also the higher threshold (the one that determines the “edge for sure”).

- Car picture requires more intensive filtering since the image is full of details.
  In the code this is implemented by using again a filter (median filter) with a kernel proportional to the size of the image. It can be clearly seen by looking at the image that even after this processing there are still many details in the image. However, it is enough to correctly extract the frame.



Example of frame detection for Hard6.png image.

In the end we can say that there is no global tuning for thresholding parameters since the gradient response depends also on how an image is noisy, how much filtering is applied or other pre-processing.

Idea for other possible approach: sharpening the image, for instance by using the Laplacian operator (actually -Laplacian) and then thresholding. In this way, edges are enhanced. The key points is to perform morphological operations (morphologyEx() function with kernel established by using the function getStructuringElement()) and only then find contours.

2- Classification (training):

Task: Develop a simple CNN with 3-4 convolutional layers and a couple of fully connected ones to perform image classification on the CIFAR10 dataset.

The code work as follow:

- **Pre-processing:** The CIFAR10 dataset is extracted from Keras repository and it is divided in training set(50000 samples) and test set (10000 samples). Images are 32x32x3 and values range between [0, 255]. Max normalization is applied to data.
- **Model creation:** The CNN architecture is established (more details after) . Then the training set is fed through the network in batches. Batch size is chosen to be 128 since it doesn't affect too much computation time and there is a gain in terms of accuracy with respect to lower batch sizes.
- **Performance evaluation:** Accuracy is evaluated for the training set along each epoch. Overall accuracy is computed for both training and test set. For test set is considered also the confusion matrix (more comments afterwards).

CNN structure:
Basic idea is to alternate convolutional and max pooling layers to capture details in the images. I tried two different strategies:

- In the first one I simply alternate a 2D convolutional layer and a Max pooling layer, which means only maximal values are retained at each conv Layer.
  This structure reduces a lot computational efforts, nevertheless it provides good results in terms of accuracy both on the training set and test set (around 70%).

- The second network instead, it is aimed to be a refinement of the previous one. Only one Max pooling layer is used every two convolutional layers. Drop out layers are added to prevent overfitting.

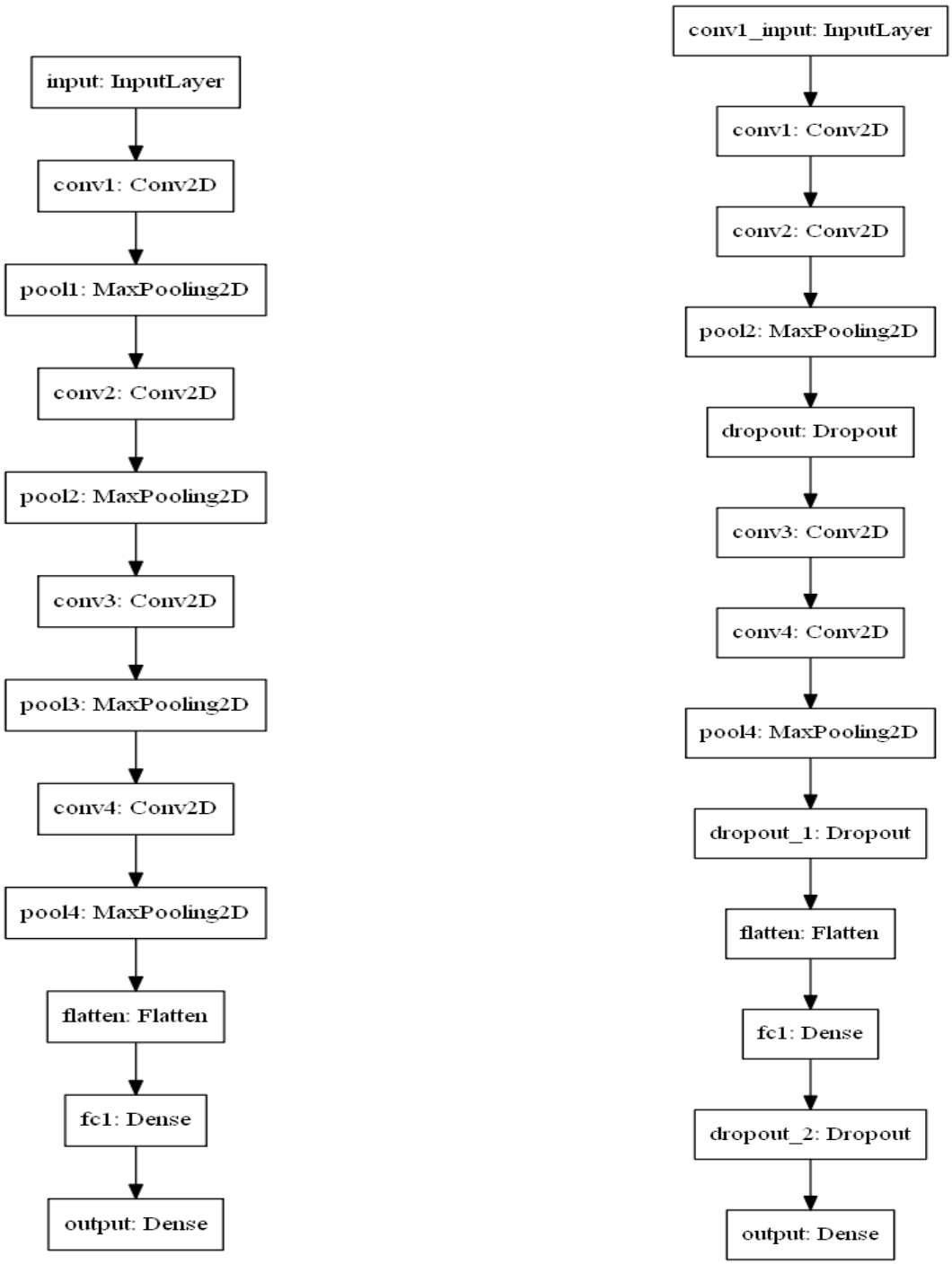


Figure: Model 1 and Model 2 architecture.

Parameters settings:

- No. of layers: not too many layers, 3-4 are enough for this dataset to provide an accuracy value of 70% on the test set. However, I noticed that even if performances were similar if considering the CIFAR dataset, while using the extracted frames, especially the ones which are not originally from CIFAR, the use of 4 layer instead of 3, improve considerably the performance.
- Kernel size: first idea was to use decreasing size at each layer to capture each time finer structures, but since the images we are dealing with are very small, I chose a fixed size equal to (3,3).
- No. of filters: same reasoning, therefore I used an increasing size (32, 32, 64, 64)

Other parameters have been set as seen in previous courses, e.g. padding = 'same' to guarantee input size equal to output size, activation function = 'relu' in order to prevent gradient vanishing … etcetera.

Two dense layers are applied in the end:

- the first one contains the final feature vector: for the first architecture I choose 64 features and for the second 128 (note that even 128 is significantly low with respect to 32x32 = 1024)

- the final layer has size 10 as the labelled classes.

To provide accuracy close to 70% with this architecture 10 epochs are enough, but I set 20 to get slightly better results. This could lead to overfitting: to prevent it I used a validation set (0.2 of the training set) and drop out approach at each layer.

Comments on confusion matrix:
From the confusion matrix we can clearly see that most common errors are between very similar classes as "Cat" and "Dogs" or "Truck" and "Automobile". They are similar in the sense that they share similar features/structures. It is expected then to have more mismatches between those classes.

```
[[818  13  33  23  27   4   6   9  44  23]
 [ 11 869   3   9   5   3   9   1  12  78]
 [ 62   3 607  66 123  43  51  26  13   6]
 [ 13   6  45 624  75 139  55  28   7   8]
 [ 10   1  22  45 825  22  28  34  10   3]
 [  8   1  35 173  61 666  12  36   2   6]
 [  3   2  39  53  67  17 802  10   6   1]
 [  9   0  22  31  88  34   3 803   3   7]
 [ 47  22  11  12   9   2   7   5 864  21]
 [ 22  44   3  19   5   3   4  13  15 872]]
```

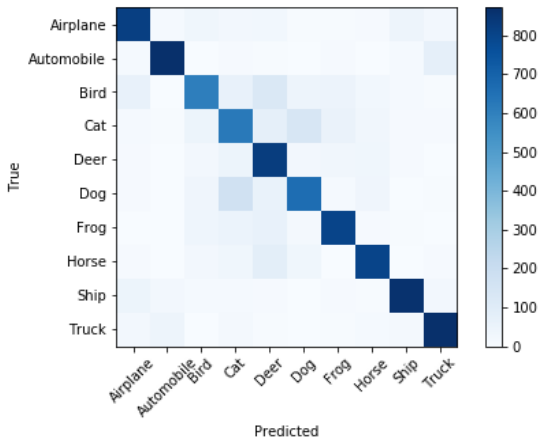


Figure: Confusion matrix.

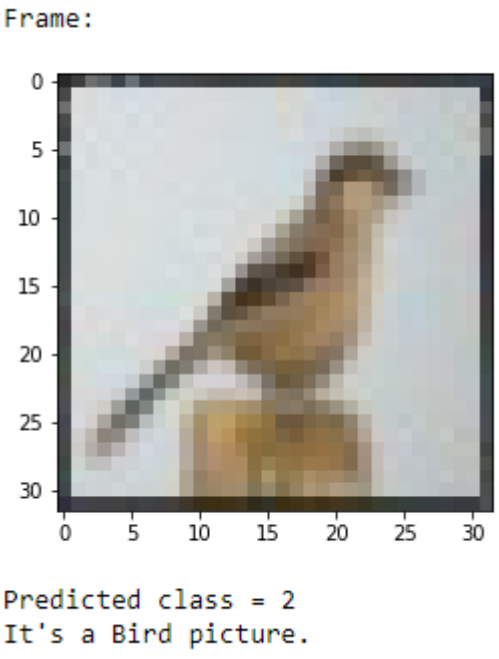
3-Classification:

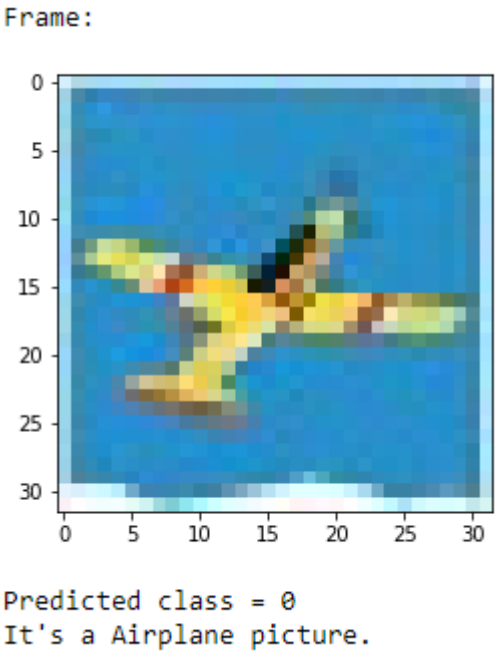
Task: Feed the content of the frames to your network to classify into one of the 10 classes in the CIFAR dataset.

Note: in the folder "Frames", that is provided together with the codes, there are the frames extracted from one execution of the code Lab5.cpp. Discard them and put in the folder new frames if you want to try with different ones.

Classification task is then performed by using as a predictive model the previously trained CNN.
Multiple classifications are done on the same picture if the extracted frames are doubled or more.

In general images from the original dataset are classified correctly.
For the other one classification become harder, especially for kittens. It could be that in the original dataset there were no kitten images without background. Also, it could be that the network is not able to capture correctly features on kittens since it is a classe with high number of mismatches.
Anyway, the main problem remains the overfitting on CIFAR dataset images, while performances on external images ones are too low.
This situation seems to happen even using a combination of different approaches (validation set, test set, drop out on neurons, regularization with max normalization, additional gaussian noise).
Surprisingly the model one which has less parameters seems to work better than the second one on the non-CIFAR images (not perfectly though).





To provide better results:

- **Change CNN user-defined parameters:** combination of different kernel size and depth. For instance, change the number of filters to (32, 32, 64, 64) in (32, 32, 64, 128) or change the strides. *(However my computer is not too powerful to perform a very large amount of different comparisons.)*
  The results in terms of accuracy on the training/test set of CIFAR doesn't seem to be affected. The

main changes were on the classification of frames.
From the trials I made, the more are the computed parameters (= the weights), the more the classification is good for the CIFAR images but at the same time the classification for the non-CIFAR images is worsened.
So user-defined parameter are need to be chosen in a way that weights to be computed are not too much.

- **Change CNN architecture:** other strategy can be exploited as for example by introducing batch normalization or by using over layers additional noise (gaussian is a common choice).

- **More pre-processing:** for instance data augmentation can be used. Rotating e scaling the images could be useful e.g. this should help especially in generalizing kittens' images.

- **Other architectures**: more advanced deep learning or machine learning techniques (RNN, Autoencoders etc.)

- **Multiple Classifier:** use multiple classifiers and then take the more common outcomes for each image.