koda: Deep Learning-Enhanced Pipeline for Keyword Detection in Document Images

Matteo Pellegrino, Federico Terzi

January 10, 2020

Abstract

TODO

1 Introduction

TODO

2 Architecture

KODA architecture was inspired by a Dropbox article [1] which defines the steps to solve a problem similar to the one this project is based on.

The process is divided into two macro tasks (shown in Fig. 1): corners detection and document analysis. The formers takes as input a generic image containing a document and returns the 2D coordinates of its corners. The latter use the corners coordinates to warp the image, apply an OCR to read the document words and returns the original and warped image with the highlighted keywords specified by the user. Considering KODA as a black box, the user loads an image and, for each keyword specified, retrieves a copy of the image containing the highlighted word.

[1] https://blogs.dropbox.com/tech/2016/08/fast-and-accurate-document-detection-for-scanning/

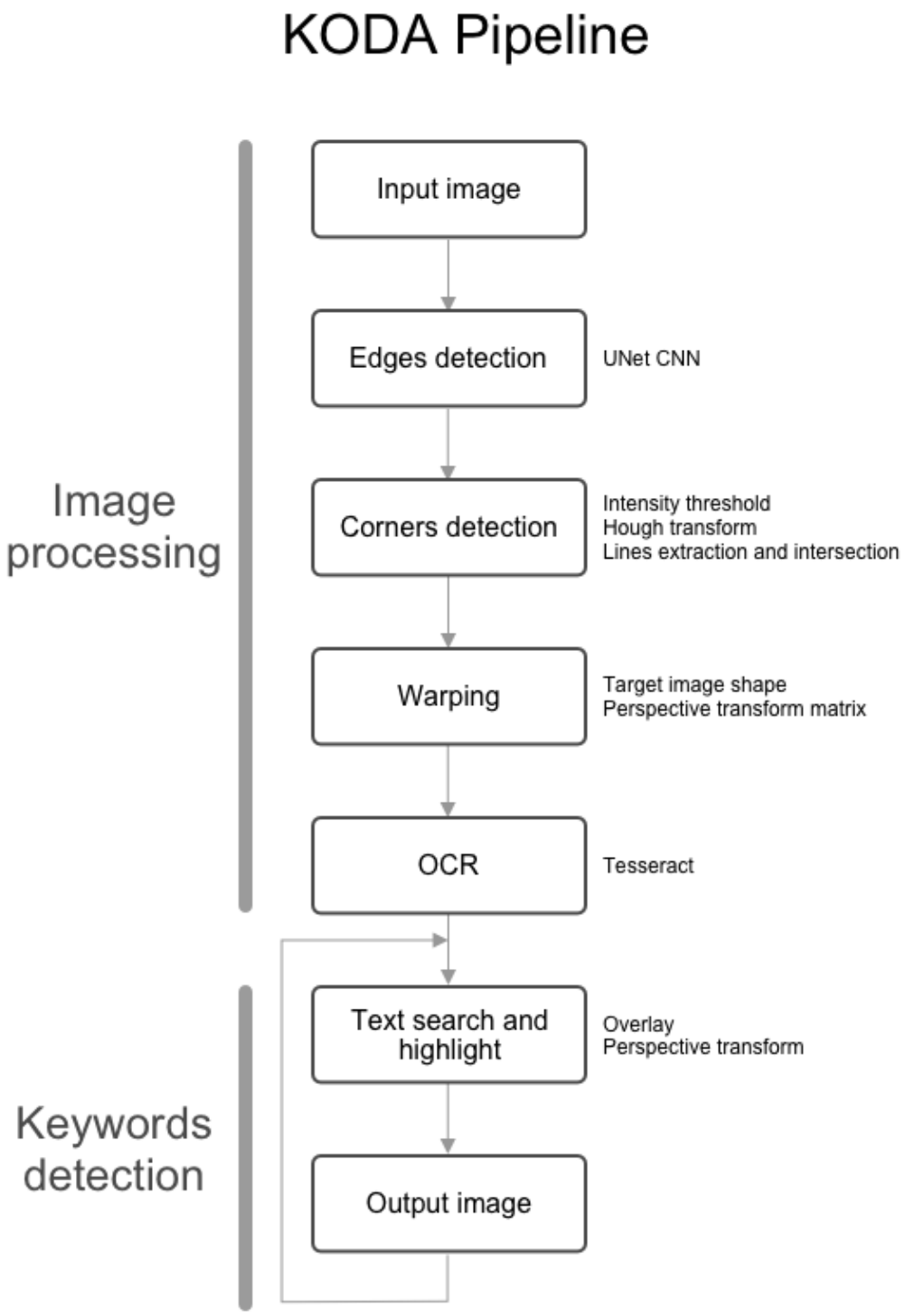

Figure 1: KODA Pipeline scheme with sub-tasks. The computationally expensive processing is done once per input image, then the user can search for keywords and retrieves the output image multiple times repeatedly.

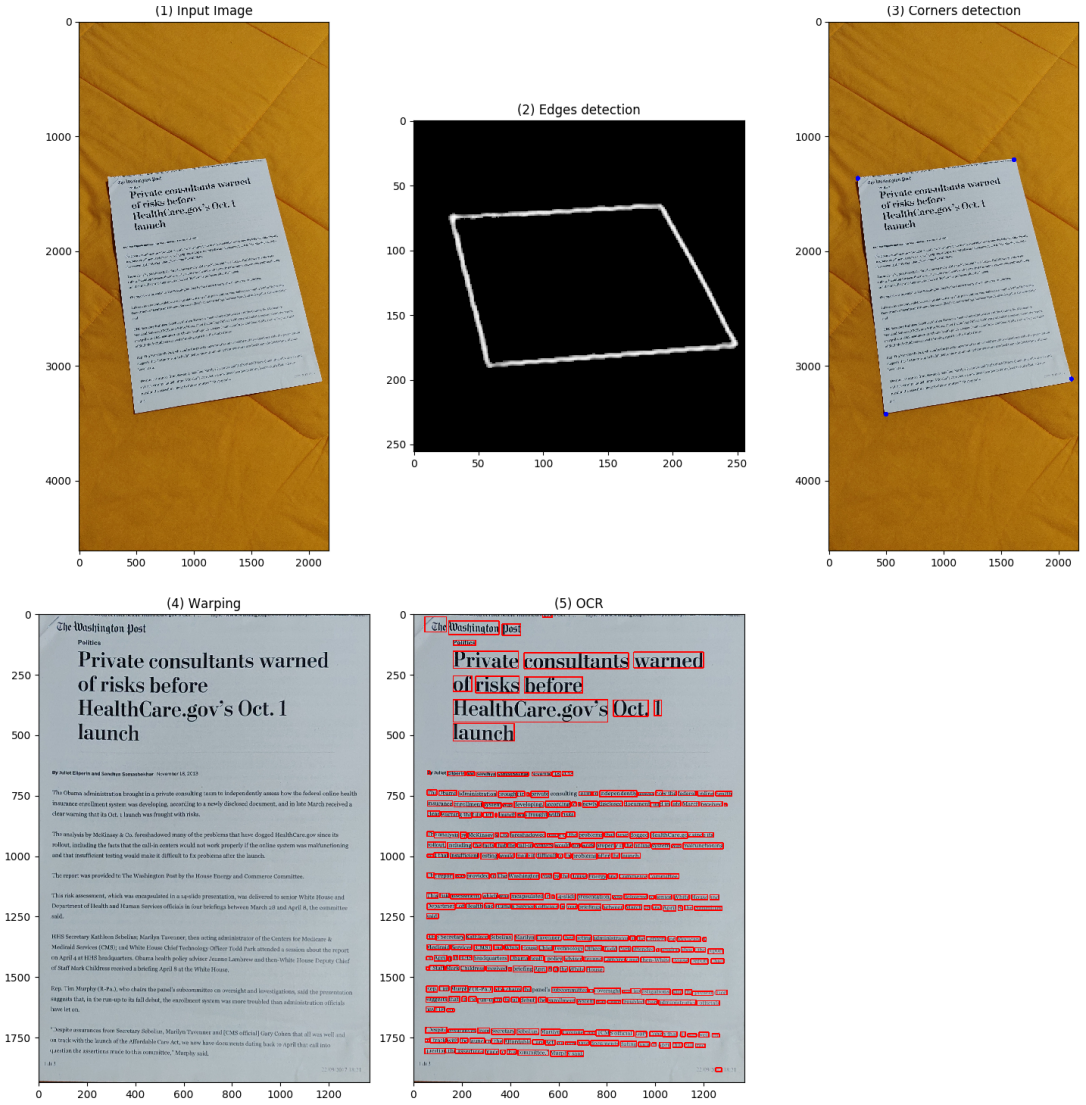

Figure 2: Image processing tasks example. Note: edges detection returns a resized (256x256)

3 Edge Detection

Above all the possible approaches, the pipeline proposed in this paper heavily relies on edges to detect documents inside images. Therefore, in order to obtain good results, it is crucial to employ a solid *edge detection* algorithm.

3.1 First Attempts with Canny

Our initial attempts to detect document edges involved the well known *Canny edge detector* [TODO REF]. Although applying the filter directly does detect the sheet edges, it presents many artifacts due to the text printed on the document itself (as seen in Fig. 3 (B)).

In order to mitigate the problem, a *Gaussian filter* [TODO REF] is applied before the edge detection, obtaining a clear highlighting of the interesting edges (as seen in Fig. 3 (C)).

Despite the result, after many observations it became clear that this approach was not robust enough to work in all real-world scenarios. In particular, Canny's parameter tuning turned out to be a major problem, as it can be seen in Fig 4, where a combination of parameters, despite producing good results in some scenarios, proved itself incapable of generalizing well in others.

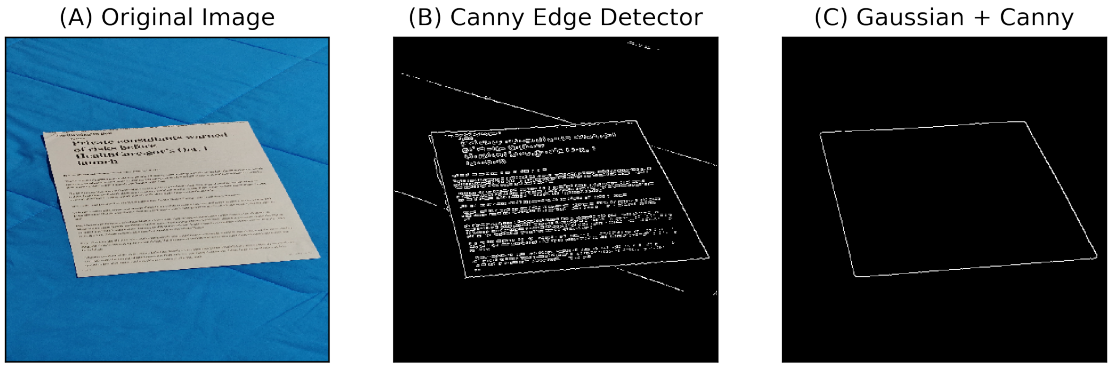

Figure 3: Application of Canny edge detector to the original image without/with a Gaussian Filter

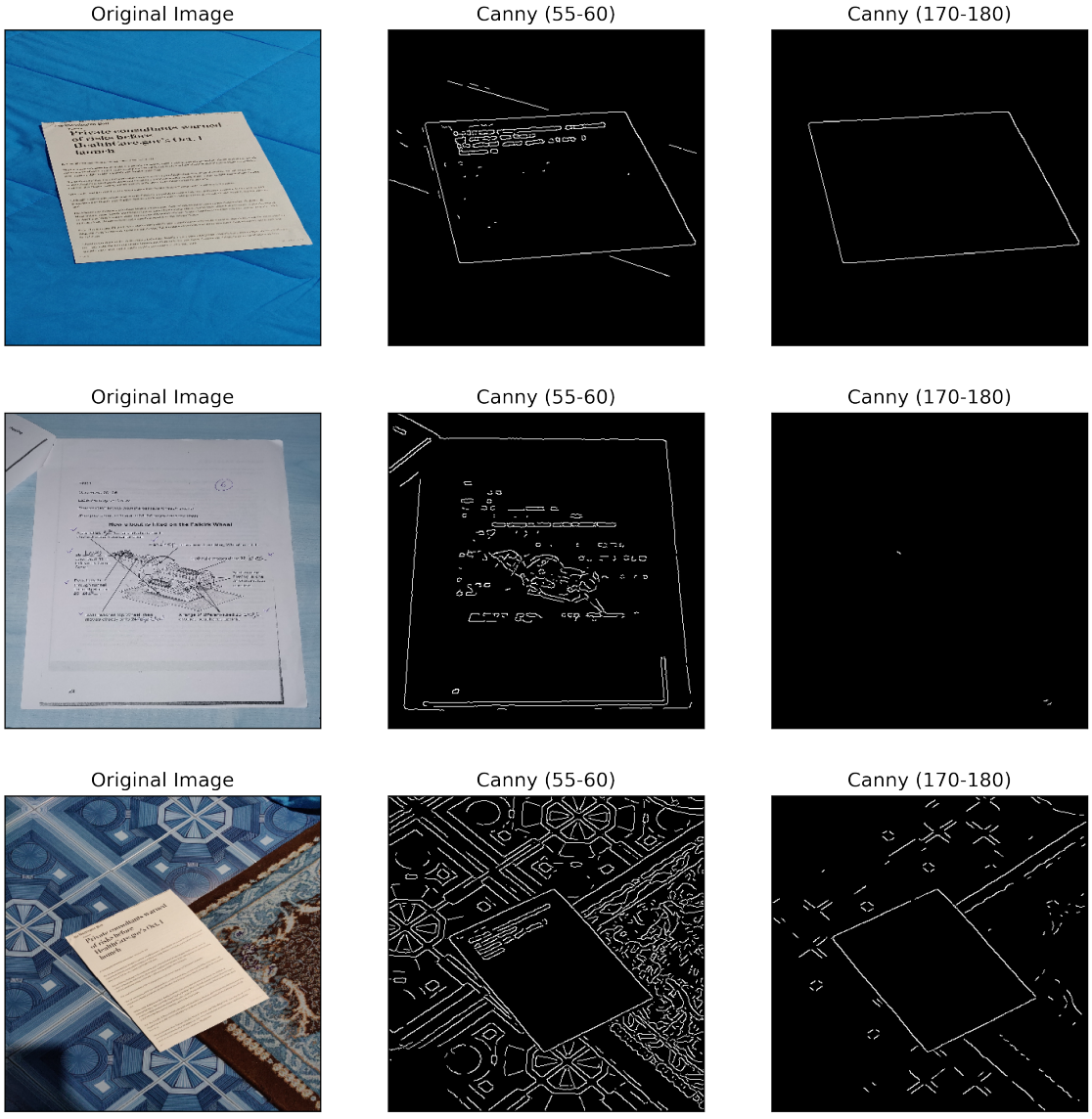

Figure 4: Comparison of different thresholds for Canny edge detector

3.2 Deep Learning Approach

In order to make the pipeline able to deal with real-world images, it is crucial to develop an edge detection mechanism capable of generalizing better in heterogeneous situations. For this reason, a *Deep Learning*-based approach was explored with the aim of creating a model capable of distinguishing document sheet edges from the rest of the image.

Due to the lack of a suitable dataset with labelled document edges, a custom one was created. In particular, 250 images containing document sheets were taken under different lighting conditions, backgrounds and positions. Thereafter, every image was labelled with the 4 corner coordinates of the document.

For this particular task, *Keras*, an open-source Python library for Deep Learning [TODO REF], was chosen to build the model. Many experiments were made to find the right architecture, most of them exploiting *convolutional neural networks*. Above all of them, KODA uses *U-Net*[TODO REF], a popular architecture for *object segmentation*, and in particular an open-source implementation for Keras [TODO REF].

The model takes a 3-dimensional input image (RGB) and outputs a 2-d map of the edges. Although the shape of those matrices can vary, KODA uses a 256 pixels side for both input and output, as it provides a reasonable compromise between size and resolution. Fig. 5 illustrates the way a sample is fed to the network. In particular, the input image is resized to a 256x256 RGB image and the output label is a gray scale map of the expected document edges.

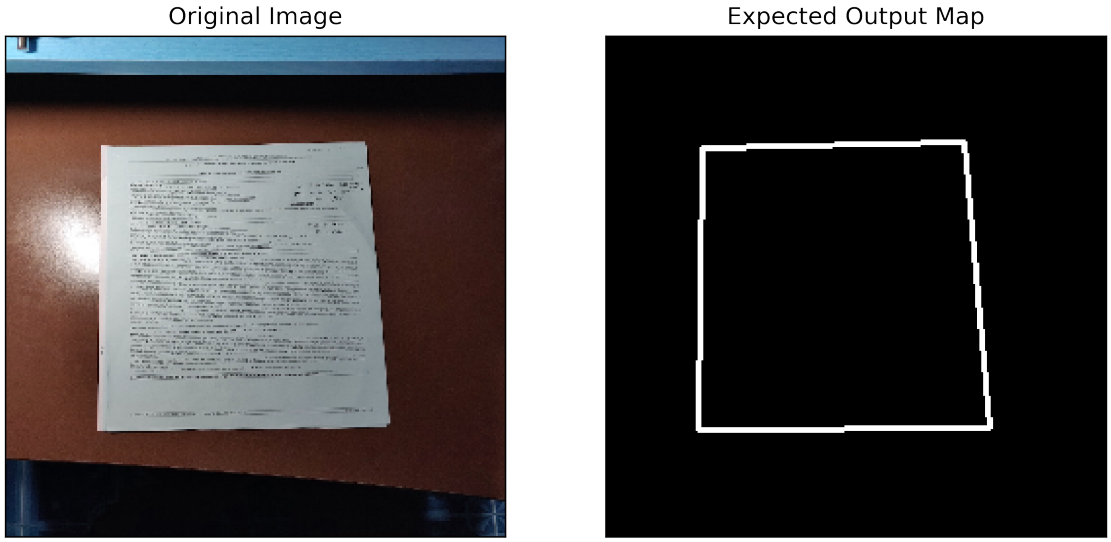

Figure 5: Example of the way a sample is fed to the network.

Due to the high number of required samples, the dataset itself is insufficient to fully train the network. For this reason, a technique called *data augmentation*[TODO REF] is applied to provide the model with enough variability. In particular, before being fed to the network, a random combination of transformations, such as scale, translation, rotation and brightness changes, is applied to the image and to the resulting edge feature map, as shown in Fig. 6.

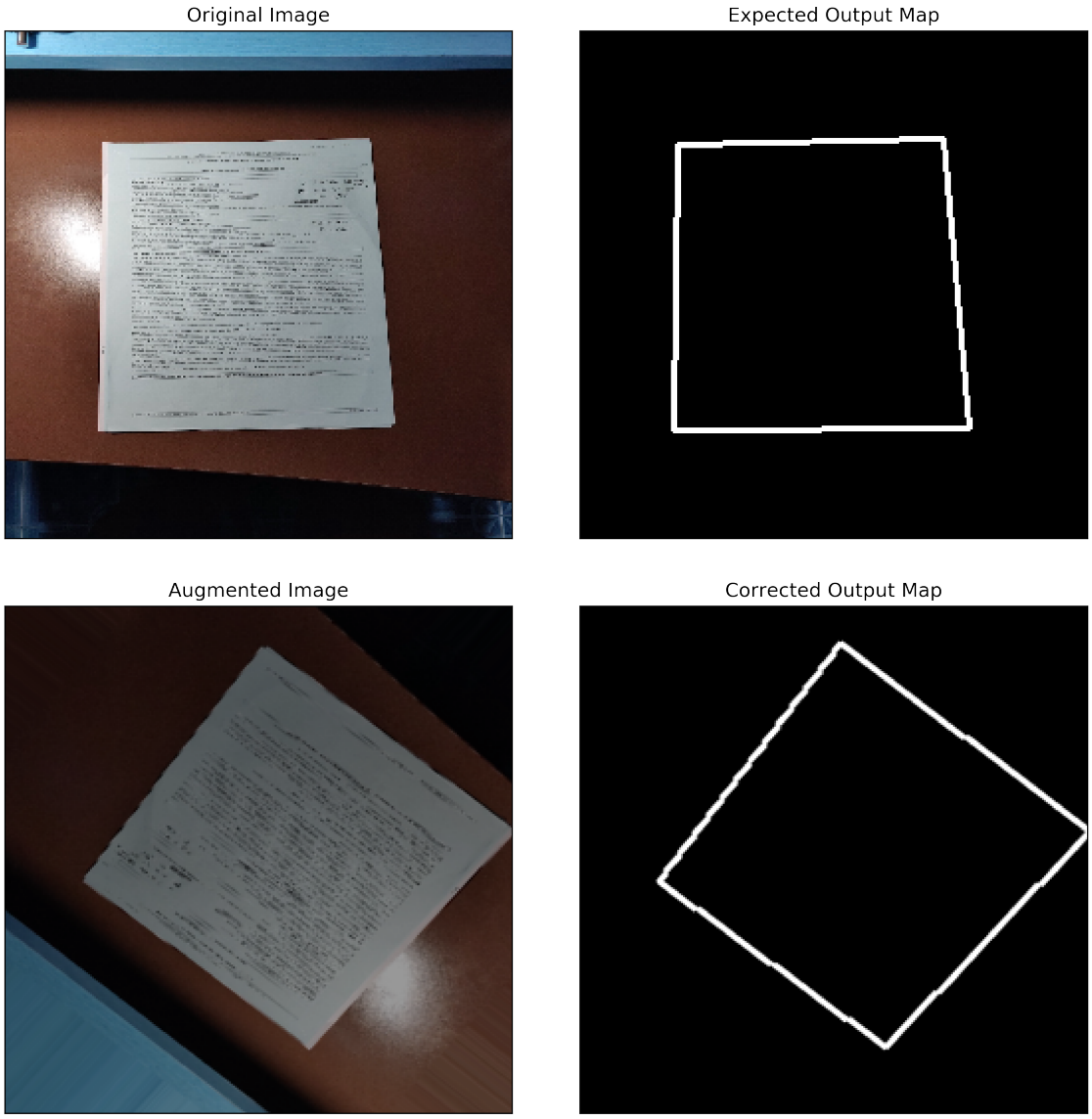

Figure 6: Example of Data Augmentation

During the training, finding the best compromise between good results and overfitting proved to be a challenging problem. In particular, the *loss function*, in this case *binary crossentropy*, was not representative of the model's quality. As it can be seen in Fig. 7, the loss value of the validation set settles around epoch 40. Despite this, further experimentation shows that the model still improves well beyond that epoch. For this reason, a new approach was needed to evaluate the performances of the model.

Despite being simple, the proposed solution is to save the current model's results over a set of test images after each epoch, as well as saving the model itself. This approach allows a simple comparative analysis, and allowed us to select the best model according to its actual results. Epoch 70 produced the best performing model from a generalization standpoint, while further epochs shown interesting but less effective trends. In particular, as you can see from Fig. 8, the model at epoch 70 is capable of detecting both (A) and (B) edges fairly well, whereas the one at epoch 140 presented an improved result in (A), removing uninteresting objects from the top portion of the image, but was almost incapable of recognizing edges in (B). This phenomenon could be explained as the model starting to empathize straight lines and discarding curved ones. Unfortunately, the latter are a common case and should be considered, therefore the first model was chosen.

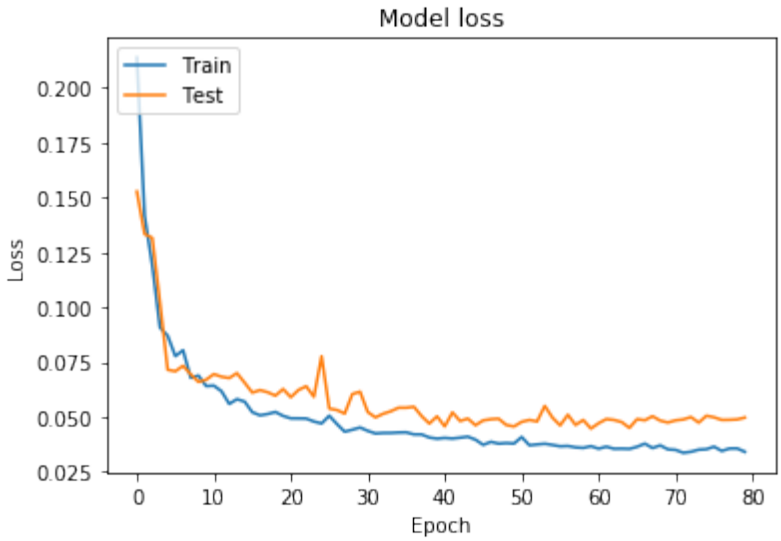

Figure 7: Model loss over the first 80 epochs.

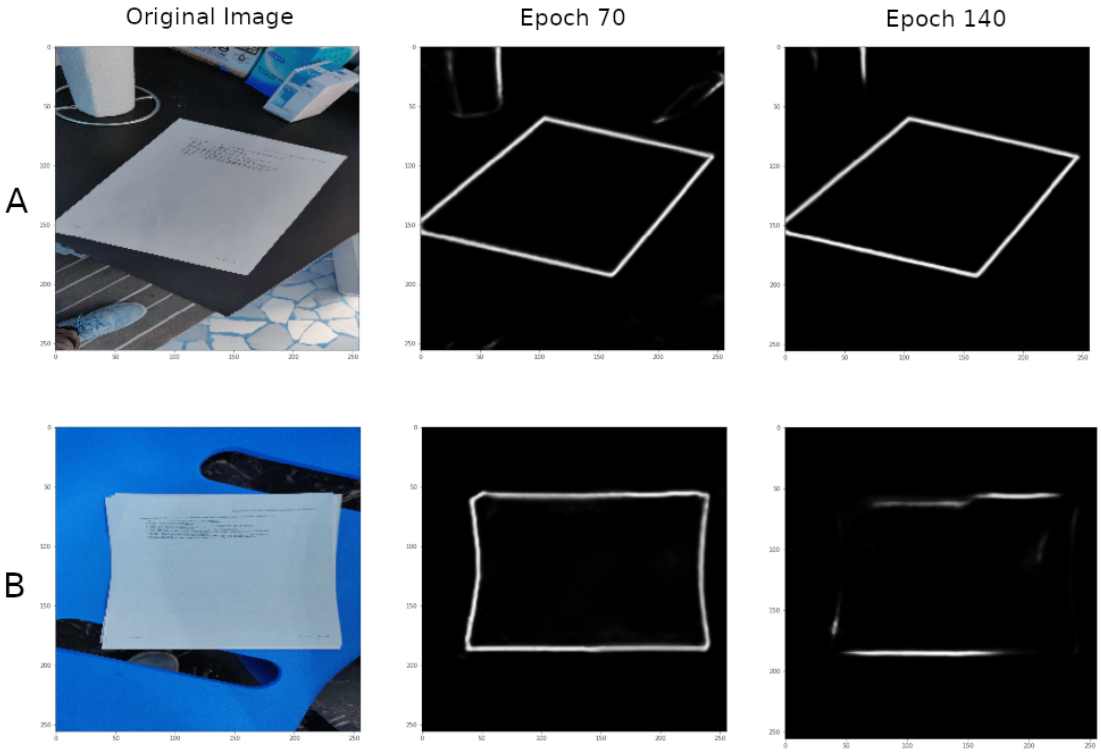

Figure 8: Comparison of the edge detector over different epochs.

An overview of the end result can be seen in Fig. 9, which also shows a comparison with the previous Canny-based approach.

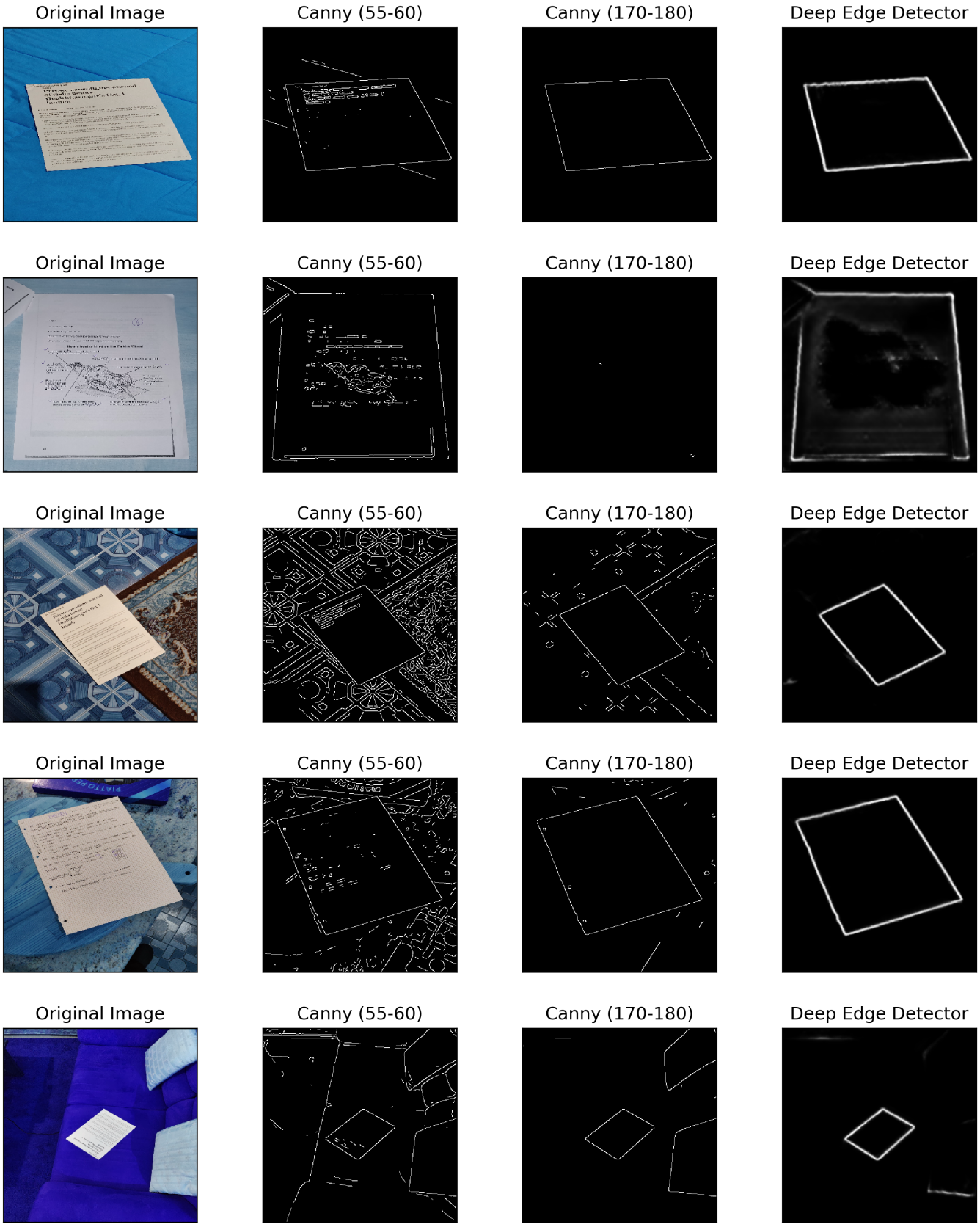

Figure 9: Comparison of Canny and Deep Learning-based edge detection.

4 Corners detection

The corners detection task goal is to find the four document corners starting from the one-channel edges image. These points should be as precise as possible in order to define the tightest convex polygon around the document. The intuitive method to solve this task is to use the Hough transformation, as suggested in a Dropbox article [2], to find straight lines that best approximate the document edges; the corners will be the intersection between those lines.

The suggested method, relies on finding more than one line per edge resulting in a set of intersections points. Those points are used to compute polygons and the one which best approximate the document edges is chosen to define the four corners. However, from empirical experiments, this could lead to an high amount of intersections meaning an enormous amount of polygons to test. Instead, KODA implementation extracts the best four lines ahead, so their intersections are immediately the best corners.

All the operations described in the following section are computed with reference to the resized edges image which is by default 256x256.

4.1 Hough Transform

OpenCV provides an interface for computing Hough lines [3] using polar coordinates (rho, theta); the problem is to calibrate the parameters and select the best lines retrieved. In order to remove unwanted noise in the edges image, an intensity threshold, which value was determined empirically (80), is applied before submitting it to the OpenCV interface. The threshold and the resolution interface parameters also were found empirically. The input images come from a natural environment with no control of lighting, so it can be expected that edges images intensity may heavily varies. For this reason, if no Hough lines are found, the task iteratively tries to reduce the threshold parameter by 10% until enough lines are found or a maximum number of iterations is reached (default is 3).

Once a set of lines is found, a sub-task is responsible to select the four best ones, one per each document edge. This is accomplished exploiting the fact that the Hough lines returned by the OpenCV interface are ordered by confidence: the first four strongest ones are selected.

[2] https://blogs.dropbox.com/tech/2016/08/fast-and-accurate-document-detection-for-scanning/
[3] https://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/hough_lines/hough_lines.html

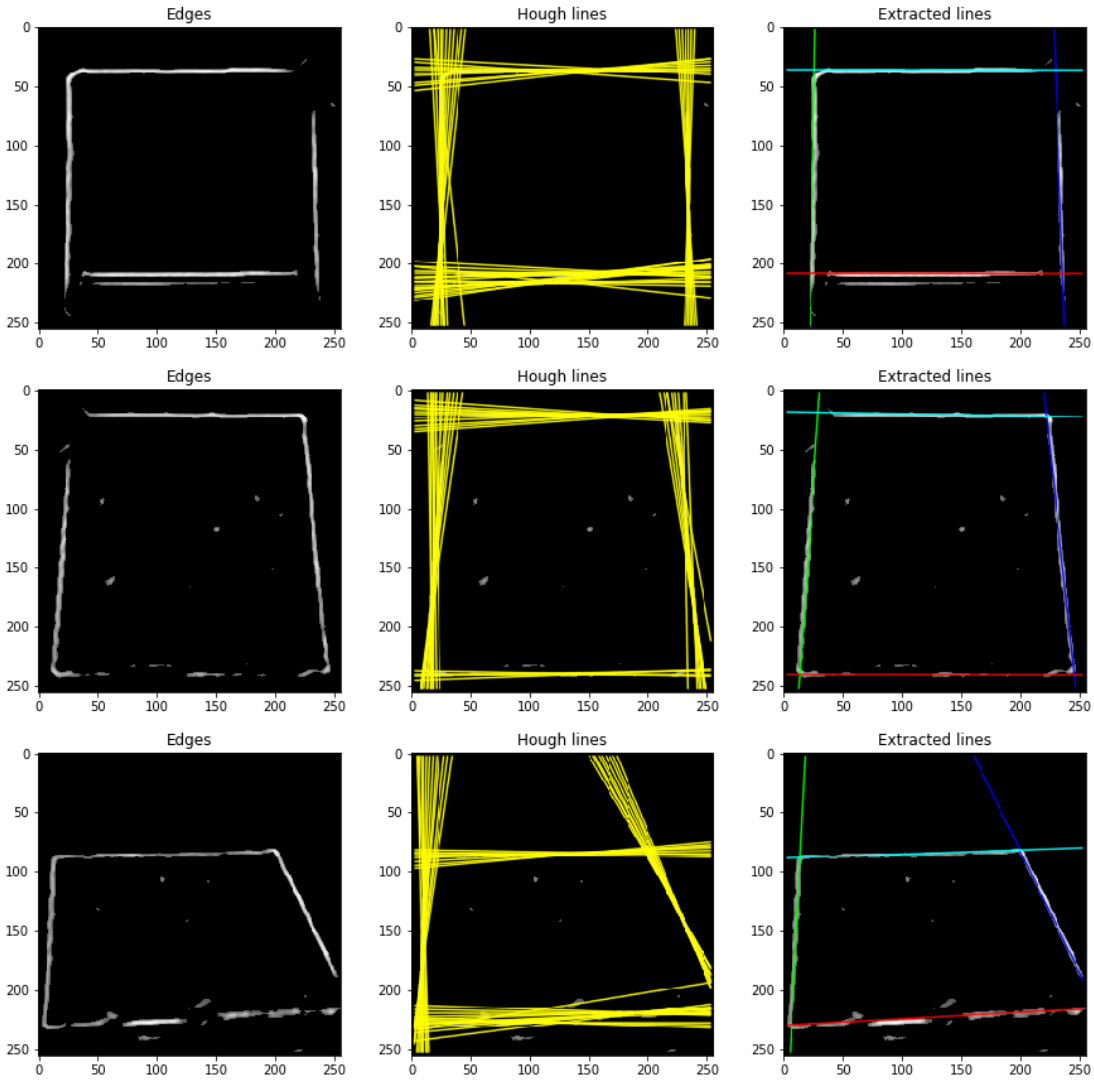

Figure 10: TODO

4.2 Computing corners

Once the four Hough lines corresponding to the document edges are found, computing the document corners means to correctly intersect the lines. They are ordered based on theta, so that parallel lines are near each other, then the intersections between alternated lines are computed. In particular, the intersection using polar is calculated as follow:

$$\begin{bmatrix} \cos(\sigma_1) & \sin(\sigma_1) \\ \cos(\sigma_2) & \sin(\sigma_2) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \gamma_1 \\ \gamma_2 \end{bmatrix}$$

$$Ax = b$$

The implementation to find the intersection relies on [4] which may raises an exception if the A matrix is singular because the computation is not possible. Whenever this happens, a least square solution, if it converges, via [5] is provided.

To maintain uniformity for the client APi, the corners are sorted following the Cartesian quadrants and they are scaled up based on the original image size provided as input.

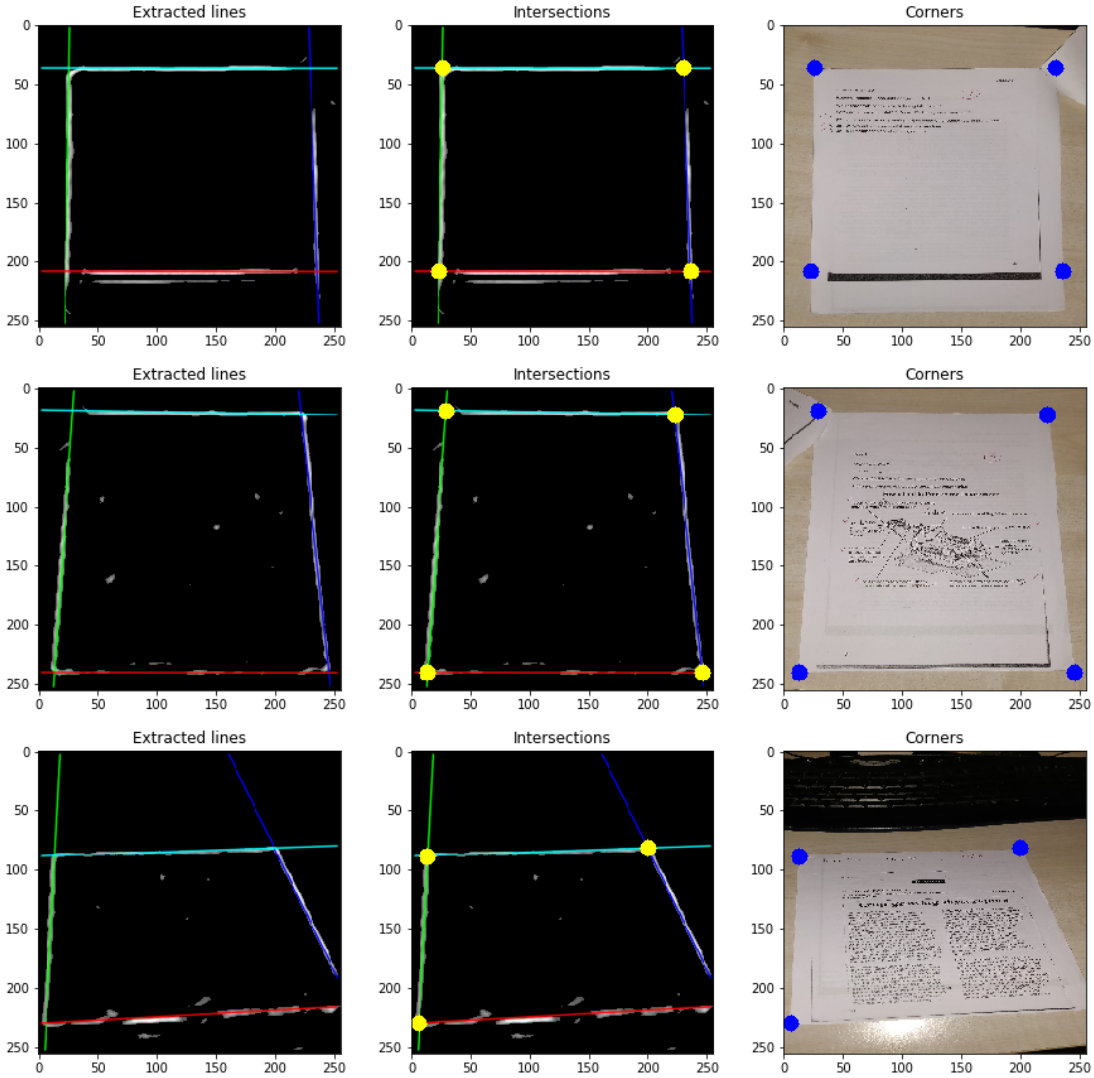

Figure 11: TODO

[4] numpy.linalg.solve: https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.solve.html
[5] numpy.linalg.lstsq: https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.lstsq.html

5 Document analysis

5.1 Warping

The first step of document analysis task is to warp the found paper within the image through a perspective transform using the detected corners. The document is expected to be A4 format, so the warped image size is computed considering the minimum of the possible widths (top, bottom) and heights (left, right), then forcing the greater dimension to be equals to the lesser one multiplied by the square root of 2 (ISO 216).

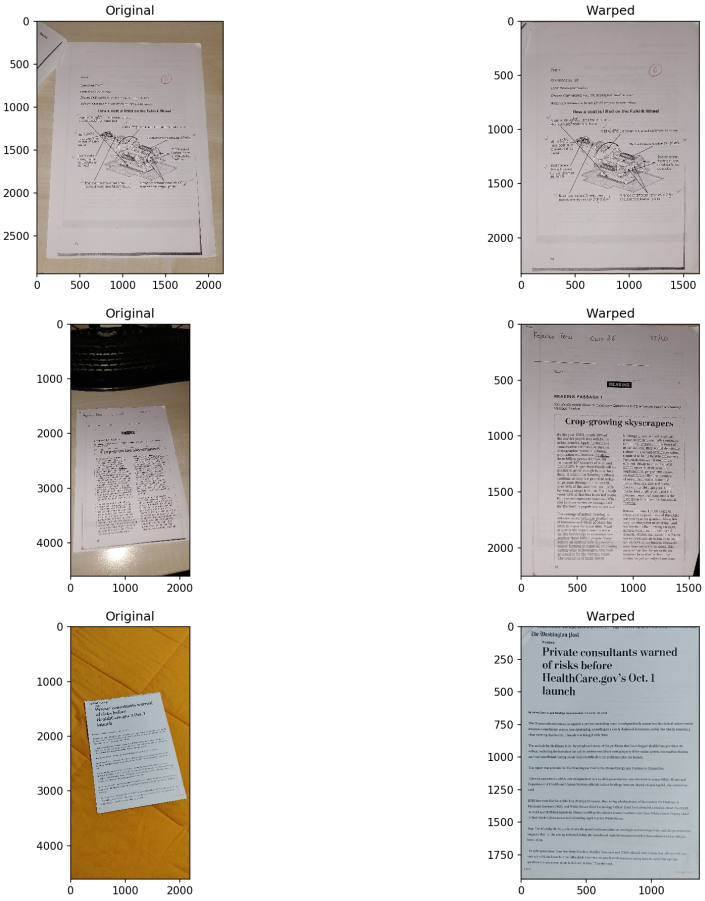

Figure 12: TODO

5.2 OCR

In order to implement the *Optical Character Recognition* feature needed to highlight keywords, KODA uses *Tesseract*[TODO REF], an open-source OCR engine by Google. Moreover, it uses *TesserOCR*[TODO REF], a Python wrapper which provides a fast binding to the C++ native library and exposes all the low-level APIs.

Due to the keyword highlighting feature, it was crucial to get the position of single words rather than blocks of text. This is easily achieved using Tesseract's *Iterators*, by setting the *Page Iterator*

Level to `WORD`. Thereafter, KODA can cycle through each word, filtering out the ones with low confidence and extracting the bounding boxes, as well as the text, as shown in Fig. 13.

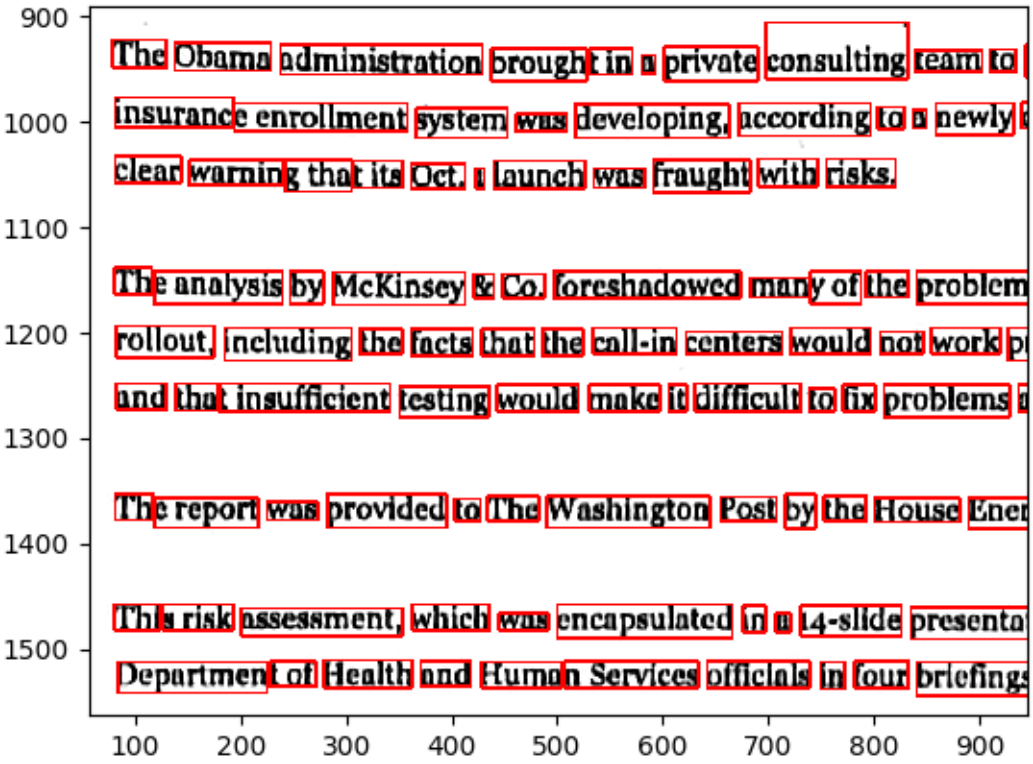

Figure 13: Example of OCR word segmenting

5.3 Keyword highlighting

The highlighting task goal is to apply a colored overlay on all the user specified keyword occurrences within the document in the image. Assuming the OCR library provides the bounding box of a detected word, a colored rectangle is drawn correspondingly on a image copy. The task output is the weighted sum of the original image and the overlay, resulting in a transparent highlighting effect on the keywords searched by the user.

The formula applied [6] is formally as follow:

$$out = src_1 \cdot \alpha + src_2 \cdot \beta + \gamma \tag{1}$$

Where:

$$\begin{gathered} src_1 = \text{overlay image} \\ src_2 = \text{original image} \\ \alpha = 0.3 \\ \beta = 1 - \alpha \\ \gamma = 0 \end{gathered}$$

However, the OCR library performs on the warped image, instead of the original one. For this reason, the overlay has to be warped back to the document shape as in the original image. This requirement is satisfied by applying a perspective transformation using the inverse of the perspective transform matrix used to find the warped image.

[6] https://docs.opencv.org/2.4/modules/core/doc/operations_on_arrays.html

The KODA implementation (shown in Fig. 14) relies on this concept, but actually transforms each bounding box instead of the entire final overlay. Transforming the warped image copy (final overlay) would leads to black pixels all over the areas the document is not present, so that adding to the original image results to darken areas outside the document. To maintain intact these areas, the highlights are drawn directly on the original image copy (not warped) after they are transformed correctly. Granted that further operations to avoid the darkened areas are possible, the KODA implementation was chosen because it does not introduce additional overhead.

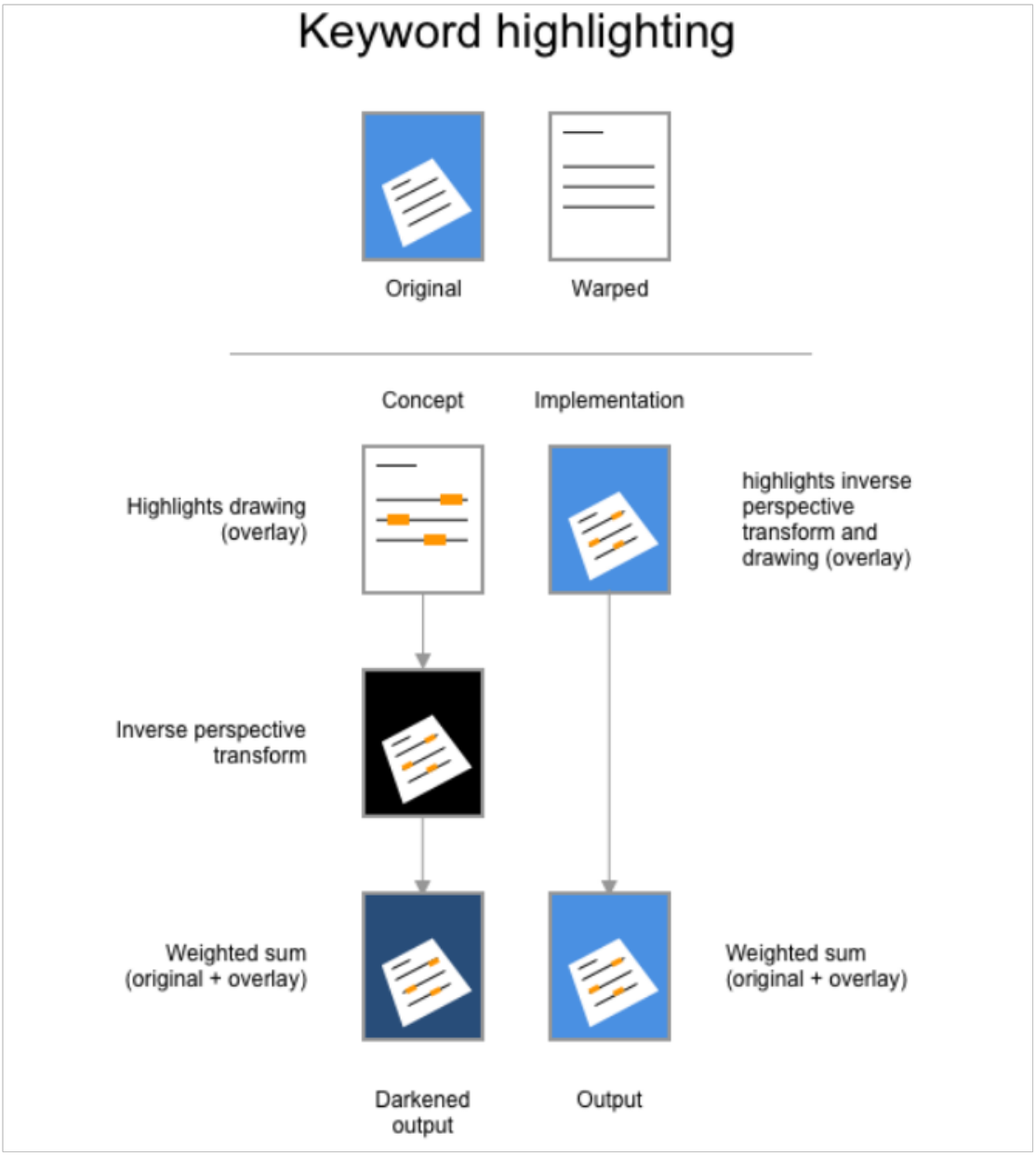

Figure 14: Process to apply the highlight overlay.

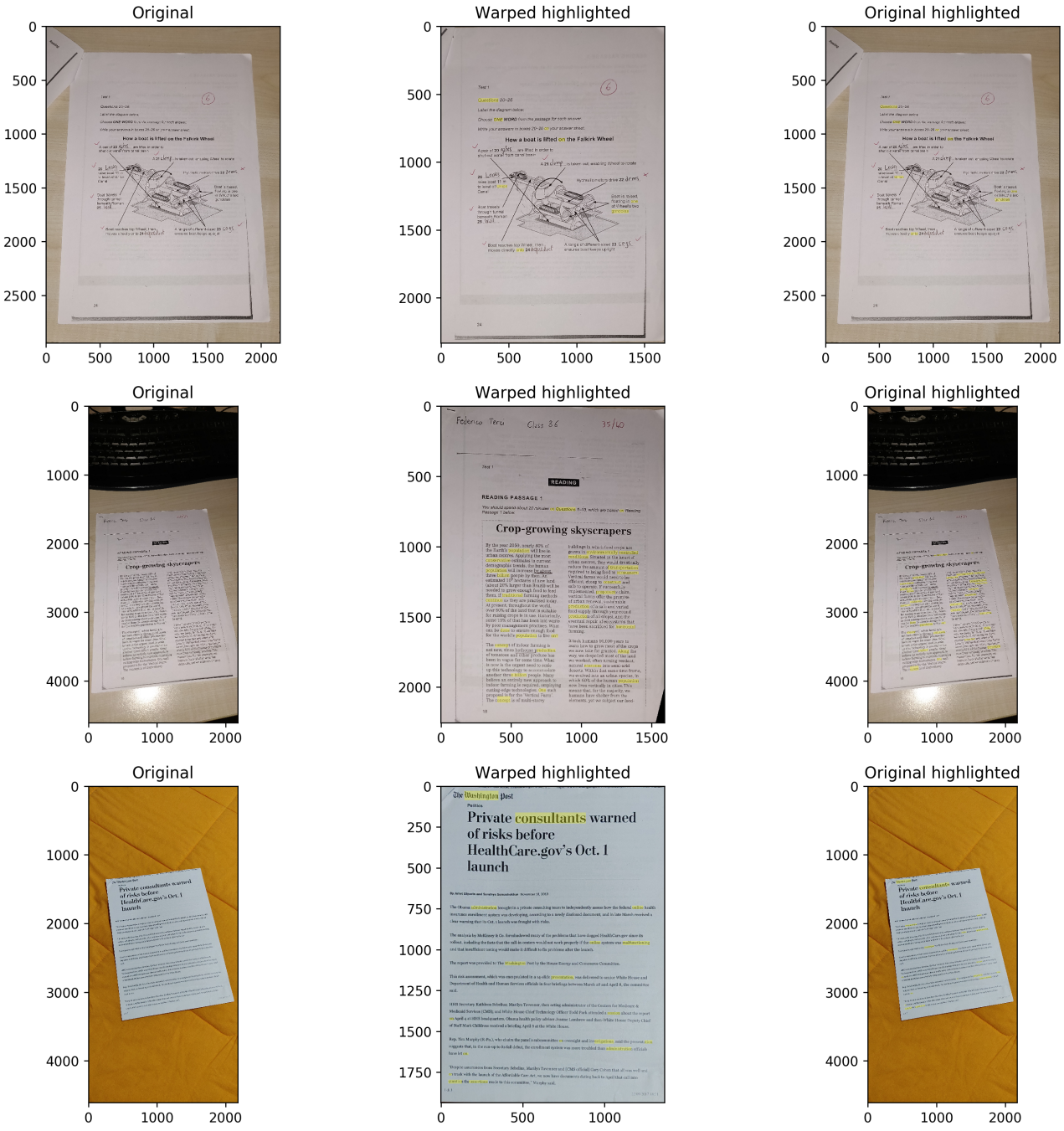

Figure 15: Examples of the keyword on highlighting. Note: the keyword is not searched as a perfect match, but as a substring of actual words. The highlight is applied to the entire word.

6 Conclusion

The proposed pipeline is able to correctly recognize and analyze the majority of document pictures in real-world scenarios, searching for the given keywords and preserving the original perspective, as shown in Fig. 16.

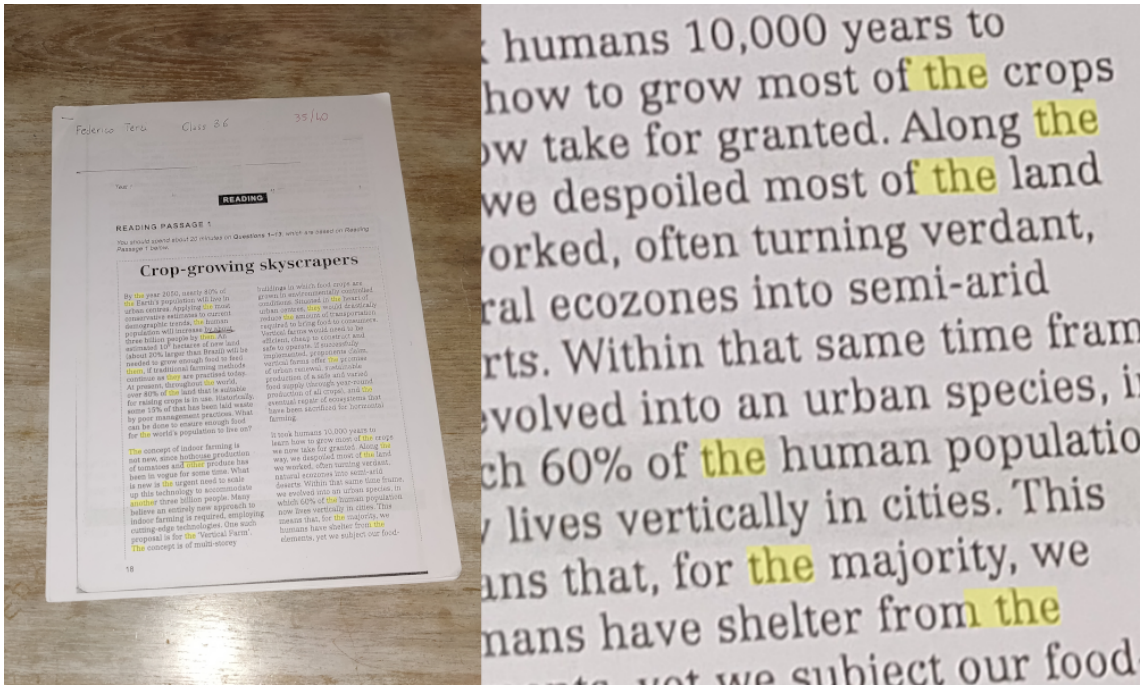

Figure 16: Example of KODA pipeline result for keyword "the". Overall output image on the left and close up on the right.

That said, KODA is still limited in certain situations. Firstly, the edge detection model was trained specifically to recognize white documents and, as a result, struggles with colored ones. Moreover, although the deep learning-based edge detection offers superior results compared to traditional techniques, it is computationally expensive. In particular, the model itself weights about 360 megabytes and takes about 0.5/1 seconds to process the image on a modern desktop CPU, which makes it unsuitable for mobile applications.

Future improvements may focus on the creation of a less computationally and memory expensive edge detection, so that the pipeline can be ported on a mobile device.