Flowchart diagram elements identification using Faster R-CNN on an augmented Online Handwritten Flowchart Dataset

Lavinia De Divitiis, Paula Mihalcea
Università degli Studi di Firenze - Scuola di Ingegneria
via di S. Marta, 3 - 50139 Firenze, Italy
lavinia.dedivitiis@stud.unifi.it, paula.mihalcea@stud.unifi.it

Abstract

In this report Faster R-CNN has to classify and retrieve elements that belongs to an image of a flowchart diagram. We will show how to build the dataset from inkml files and how to augment train set images. Finally, we will show the differences in performance on test set, between the use of data augmentation and not, on train set.

1. Introduction

The use of digital pens for acquiring text is growing ever more popular, thanks to a variety of affordable devices and technologies. While text recognition has enjoyed extensive research and achieved considerable results, the same technique tends to fail when applied to hybrid content such as flowcharts. This project, by following previous work [13][19] on such diagrams, aims to experiment with and possibly get an accurate recognition of flowcharts drawn by using the Neo Smartpen M1 device [8].

2. Faster R-CNN

2.1. Background

Faster R-CNN was originally published in NIPS 2015. Everything started with "*Rich feature hierarchies for accurate object detection and semantic segmentation*"[15] (R-CNN) in 2014, which used an algorithm called Selective Search to propose possible regions of interest and a standard Convolutional Neural Network (CNN) to classify and adjust them. It quickly evolved into Fast R-CNN[14], published in early 2015, where a technique called Region of Interest Pooling allowed for sharing expensive computations and made the model much faster. Finally came Faster R-CNN[18].

2.2. Network Architecture

We'll start with a high level overview in Figure 1, it all begins with an image, from which we want to obtain:

- a list of bounding boxes
- a label assigned to each bounding box
- a probability for each label and bounding box

The input images are represented as height x width x depth tensors (multidimensional arrays), which are passed through a pre-trained CNN up until an intermediate layer, ending up with a convolutional feature map. This is used as a feature extractor for the next part.

Next, we have a Region Proposal Network (RPN). Using features that CNN computed, RPN is used to find a number of regions (bounding boxes), which may contain objects. RPN uses anchors to find bounding boxes in the image.

Anchors are fixed sized reference bounding boxes which are placed uniformly throughout the original image, Figure 2. Bounding boxes have rectangular shape and can come in different sizes and aspect ratios.

RPN takes all the reference boxes (anchors) and outputs a set of good proposals for objects. It does this by having two different outputs for each of the anchors. The first one is the probability that an anchor is an object. The second output is the bounding box regression for adjusting the anchors to better fit the object it is predicting.

The RPN is implemented in a fully convolutional way, using the convolutional feature map returned by the base network as an input.

For the classification layer, it takes all the anchors and puts them into two different categories: background (not an object) and foreground (an actual object). In particular, those that overlap a ground-truth object with an Intersection over Union (IoU) bigger than 0.5 are considered "foreground" and those that don't overlap any ground truth object or have less than 0.1 IoU with ground-truth objects are considered "background".

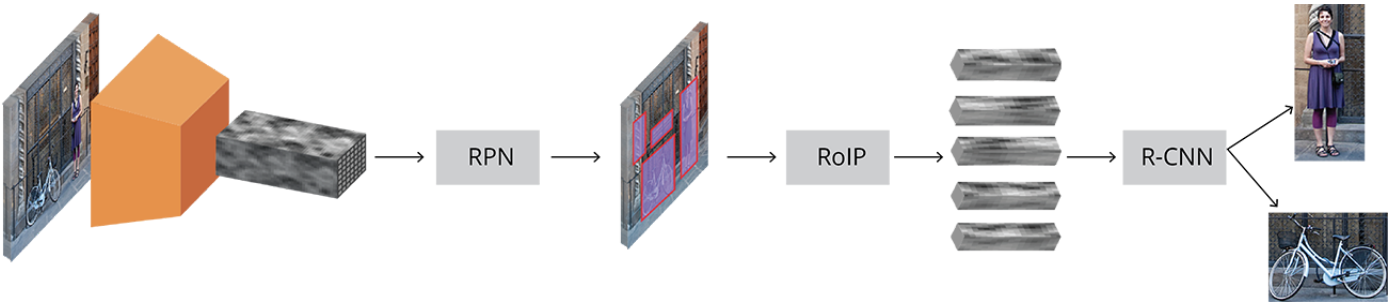

Figure 1. Faster R-CNN architecture.

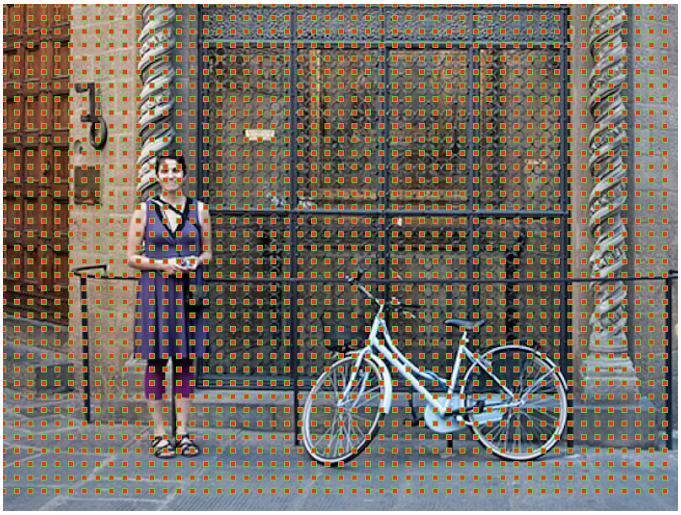

Figure 2. Anchor centers through the original image.

Since anchors usually overlap, proposals end up overlapping over the same object. To solve the issue of duplicate proposals it uses an algorithmic approach called Non-Maximum Suppression (NMS).

NMS takes the list of proposals and iterate over it discarding those proposals that have an IoU larger than some predefined threshold.

After the RPN step, we have a bunch of proposed regions with different sizes with no class assigned to them. Different sized regions means different sized CNN feature maps. It's not easy to make an efficient structure to work on features with different sizes. Region of Interest Pooling can simplify the problem by reducing the feature maps into the same size. Region of Interest (ROI) pooling is used for utilising single feature map for all the proposals generated by RPN in a single pass. In particular, it produces the fixed-size feature maps from non-uniform inputs by doing max-pooling on the inputs. For every Region of Interest from the input list, it takes a section of the input feature map that corresponds to it and scales it to some pre-defined size. The scaling is done by:

- Dividing the region proposal into equal-sized sections (the number of which is the same as the dimension of the output)
- Finding the largest value in each section
- Copying these max values to the output buffer

Region-based Convolutional Neural Network (R-CNN) Figure 3 is the final step in Faster R-CNN's pipeline. The R-CNN module classifies the content in the bounding box (or discard it, using "background" as a label) and adjusts the bounding box coordinates according to the predicted class. R-CNN takes the feature map for each proposal, flattens it and uses two fully-connected layers of size 4096 with ReLU activation, Figure 3.

R-CNN takes the proposals and the ground-truth boxes, and calculates the IoU between them. Those proposals that have IoU greater than 0.5 with any ground truth box get assigned to that ground truth. Those that have between 0.1 and 0.5 get labeled as background. It ignores proposals without any intersection. This is because at this stage, network is assuming that it has good proposals.
Bounding box regressions are calculated as the offset between the proposal and its corresponding ground-truth box, only for those proposals that have been assigned a class based on the IoU threshold.

Similar to the RPN, we end up with a bunch of objects with classes assigned which need further processing before returning them. In order to apply the bounding box adjustments the network has to take into account which is the class with the highest probability for that proposal. It also has to ignore those proposals that have the background class as the one with the highest probability.
After getting the final objects and ignoring those predicted as background, Faster R-CNN apply Non-Maximum Suppression, as explained above.

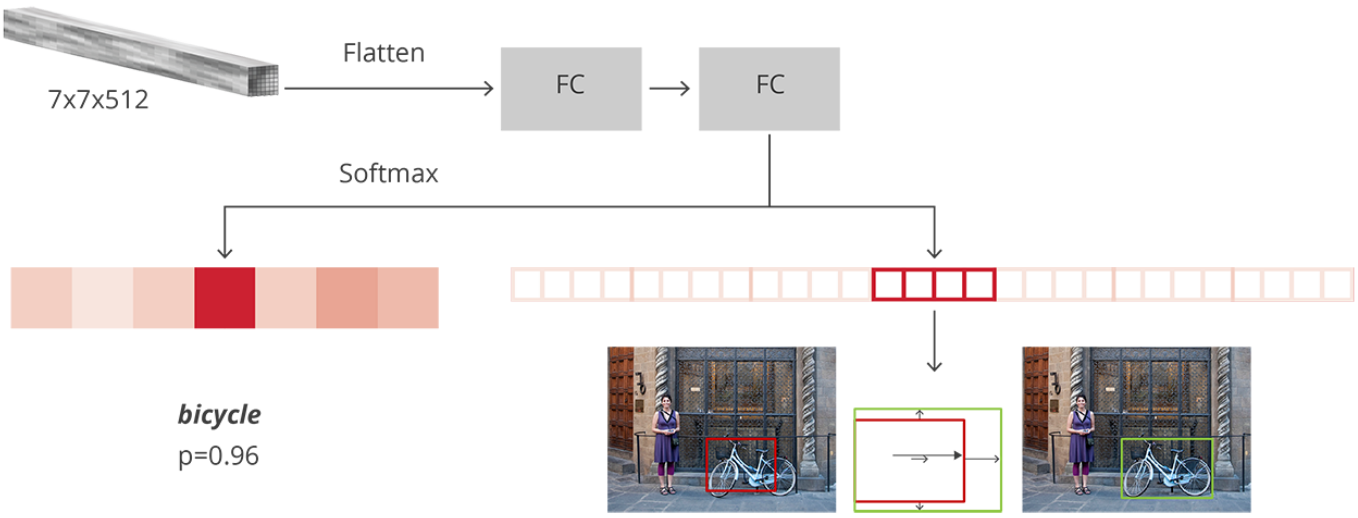

Figure 3. R-CNN architecture.

2.3. Training and implementation specifications

Faster R-CNN is implemented in Keras [3] [12]. As CNN Backbone Network we use VGG-16. We train Faster R-CNN using Adam optimizer with learning rate of 1^{-5}. The length of each epoch is the train set size.

Faster R-CNN was firstly train on a train set size of 248 without data augmentation and then on a train set size of 496 whose set also includes augmented and not augmented images. The test set size was 171 flowchart images.

Our environment is Google's Colaboratory[2] with a GPU acceleration for training. Colab resources are not guaranteed and not unlimited, and the usage limits sometimes fluctuate. This means that overall usage limits as well as idle timeout periods, maximum VM lifetime, GPU types available, and other factors vary over time. In addition, users who use Colab for long-running computations, or users who have recently used more resources in Colab, are more likely to run into usage limits and have their access to GPUs and TPUs temporarily restricted[1]. Colab's time limitation for every session, was a limitation for us, this led us to frequently reconnect the server and load the weights to keep train the model. This limitation has also increased the network's training time.

Images are resized to width 300px and height 400px. The input data is from a .txt file, one for test and one for train, which contains a set of images with their bounding boxes information. Each row, in .txt file, has the following format: `file_path,x1,y1,x2,y2,class_name`, where `file_path` is the absolute path of an image of the dataset, `(x1,y1)` and `(x2,y2)` represent the top left and bottom right real coordinates of the original image, `class_name` is the class name of the current bounding box.
Faster R-CNN has several hyperparameters to be tuned, below you will find the setup we choose for our network.
In Region Proposal Network, given anchors, we need to extract a number of boxes as the region of interests and pass them to the classifier layer. The initial status for each anchor is 'negative'. Then, we set the anchor to positive if the IOU is is less than 0.7. If the IOU is greater than 0.3 and less than 0.7, it is ambiguous and not included in the objective. It is very important to be cautious with the IoU threshold. Too low and we may end up missing proposals for objects; too high and we could end up with too many proposals for the same object.
Then, we use non-max-suppression with 0.1 threshold value. We choose 0.1 as threshold because we notice that Faster R-CNN performs better than using higher values.
Regarding RoI Pooling layer that produces the fixed-size feature maps from non-uniform inputs by doing max-pooling on the inputs, we decide to use the default number of ROIs to be processed in one time, that is 4.

3. Dataset

The project uses the Online Handwritten Flowchart Dataset (OHFCD)[13], which consists of 419 InkML files of hand drawn flowchart diagrams. This is an XML data format for representing digital ink data input with an electronic pen or stylus [20].

All Python scripts written in order to manipulate the original InkML dataset are available at this project's GitHub repository[16].

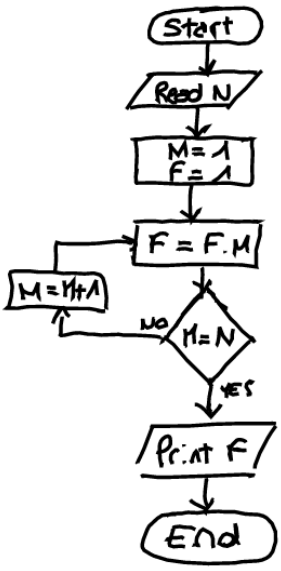

Figure 4. A file from the OHFCD dataset, as plotted by the CROHME InkML Viewer[17].

3.1. InkML file format

A file of the dataset consists of around one hundred and fifty *traces* (fig. 5), each having a unique id and describing a single stroke of the pen by using a series of (x, y) coordinates identifying successive points in said stroke.

```
<trace id="0">
1268.06 3.73599, 1268.06 3.73599, 1273.66 0.622665, 1269.3 0,
1240.04 3.11332, 1213.26 4.35865, 1197.7 3.11332, 1184.62
5.60398, 1113.01 12.4533, 1100.56 16.812, 1086.86 18.0573,
1061.33 27.3973, 1043.28 42.3412, 1032.07 67.2478, 1027.71
84.0598, 1027.71 90.2864, 1027.71 95.2677, 1029.58 107.098,
1045.14 120.797, 1057.6 125.778, 1071.92 127.646, 1193.34
123.91, 1199.56 124.533, 1312.89 122.042, 1347.76 128.892,
1383.87 127.024, 1405.04 117.061, 1421.23 98.3811, 1431.82
69.7385, 1430.57 47.9452, 1416.25 19.9253, 1398.19 6.84931,
1382 2.49066, 1368.93 0.622665, 1287.36 3.11332, 1245.02
6.84931, 1238.79 8.09464, 1222.6 11.8306, 1214.51 12.4533,
1210.77 10.5853
</trace>
<trace id="1">
1134.18 56.6625, 1134.18 56.6625, 1134.18 49.1905, 1132.94
46.6999, 1129.83 44.8319, 1111.15 43.5865, 1099.32 48.5679,
1091.22 57.9078, 1089.98 62.8892, 1116.75 69.1158, 1131.07
```

Figure 5. Trace samples from the `writer1_1.inkml` dataset file.

A series of annotations after the last trace (fig. 6) separates the strokes in semantic groups (by listing their ids), each identifying a single figure or text block in the flowchart diagram (e.g. a rectangle will form a group while the text inside of it will constitute a different group). Each group also has one out of seven possible labels: *arrow*, *connection*, *data*, *decision*, *process*, *terminator* and *text*.

```
<traceGroup xml:id="106">
    <annotation type="truth">From ITF</annotation>
    <traceGroup xml:id="107">
        <annotation type="truth">terminator</annotation>
        <traceView traceDataRef="0"/>
        <annotationXML href="terminator_1"/>
    </traceGroup>
    <traceGroup xml:id="108">
        <annotation type="truth">text</annotation>
        <traceView traceDataRef="1"/>
        <traceView traceDataRef="2"/>
        <traceView traceDataRef="3"/>
        <traceView traceDataRef="4"/>
        <traceView traceDataRef="5"/>
        <traceView traceDataRef="6"/>
        <traceView traceDataRef="7"/>
        <annotationXML href="Start_1"/>
    </traceGroup>
```

Figure 6. Trace groups example as seen in a InkML dataset file.

Another set of annotations at the beginning of the file specifies the order of the trace groups in the diagram by listing its nodes (e.g. a terminator and its text will be the first nodes in the diagram, usually followed by an arrow and a rectangle or parallelogram), including a transcription of the text traces (fig. 7); again, each node has a unique id that is later referenced in the trace groups.

```
<ink xmlns="http://www.w3.org/2003/InkML">
<traceFormat>
<channel name="X" type="decimal"/>
<channel name="Y" type="decimal"/>
</traceFormat>
<annotation type="UI">2010_IRCCyN_FC_writer1_1</annotation>
<annotation type="copyright">LUNAM/IRCCyN</annotation>
<annotation type="writer">writer1</annotation>
<annotation type="truth">"1"</annotation>
<annotationXML type="truth" encoding="Content-FlowchartML">
    <flowchart xmlns='LUNAM/IRCCyN/FlowchartML'>
        <node xml:id="terminator_1" type="terminator">
            <text xml:id="Start_1">Start</text>
        </node>
        <node xml:id="data_1" type="data">
            <text xml:id="Read N_1">Read N</text>
        </node>
```

Figure 7. Node samples as seen in a InkML dataset file.

3.2. SVG to InkML conversion

The NeoSmartpen M1 Android app[1] allows the user to save the notes in many different formats; the SVG option is the most viable choice for converting them to an InkML file, given the many similarities between these formats and, more importantly, their vector nature. A script (`svg2inkml.py`) has been written to convert SVG files created by the pen to simple InkML files containing only text traces, which can be plotted using `plot_inkml.py`[2].

[1] The iOS and Windows versions have not been tested.

[2] The plotting of such files is only possible after having removed the parsing of annotations in both `parse_inkml.py` and `plot_inkml.py`, since annotations must be added by hand in the converted SVG file and are most likely absent unless the user has taken

The `NeoSmartpenM1_demo` folder in the project repository contains some InkML files generated using this program, while its twin folder `NeoSmartpenM1_demo_png` has the resulting PNG images, as shown in fig. 8.

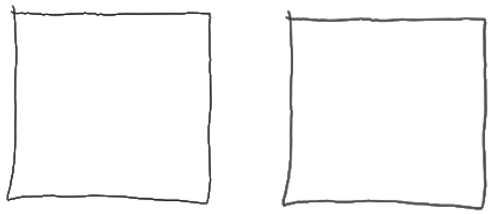

Figure 8. Comparison of a SVG image obtained from the Neo Smartpen M1 device (left) and its equivalent after converting it to InkML (right).

3.3. InkML parsing

Even though the InkML format has been published by the World Wide Web Consortium (W3C) in September 2011[20] and is very similar to the SVG format, it does not have extensive support and for the purposes of this project an ad hoc Python parser has been created in order to correctly process the dataset.

The `parse_inkml.py` script, written in the form of a function[3], takes an InkML file as input and returns a Pandas dataframe[9] containing the complete list of traces, each with its own (x, y) coordinates in the original InkML reference system and the trace group they belong to, as well as that group's class. The program works by extracting trace lines from the file and processing them in a suitable manner by using native Python string transform functions (e.g. `split()` and `replace()`); it then reopens the file to parse annotations and assign a unique group id to each trace group (note that this id will be different from the original one, as it needs to start from 0).

3.4. InkML visualization

The dataframe obtained from the parsing of an InkML file can be passed as input to the `plot_inkml.py` script, which uses the matplotlib[7] library to plot points of the traces extracted from the original InkML file and thus view the actual flowchart diagram contained in each file[4] (fig. 10). It also allows the user to save the generated plot as a 640x480 pixels PNG image and eventually highlight each trace group with a different color based on its class (see fig. 11)[5].

some time to write them – causing the script to raise an error when looking for them.

[3] As almost every other script in the repository of this project.

[4] An alternative viewer has been used for debugging, see [17]

[5] Colors can be defined in the `colors` file.

	trace	class	group_id
0	[[1268.06, -3.73599], [1268.06, -3.73599], [12...	Terminator	0
1	[[1134.18, -56.6625], [1134.18, -56.6625], [11...	Text	1
2	[[1173.41, -94.6451], [1173.41, -94.6451], [11...	Text	1
3	[[1157.22, -70.9838], [1157.22, -70.9838], [11...	Text	1
4	[[1212.02, -57.9078], [1212.02, -57.9078], [12...	Text	1
5	[[1258.09, -62.2665], [1258.09, -62.2665], [12...	Text	1
6	[[1311.64, -89.0411], [1311.64, -89.0411], [13...	Text	1
7	[[1292.34, -56.6625], [1292.34, -56.6625], [13...	Text	1
8	[[1236.3, -244.707], [1236.3, -244.707], [1236...	Arrow	2
9	[[1401.93, -235.367], [1401.93, -235.367], [14...	Data	3
10	[[1307.91, -363.636], [1307.91, -363.636], [13...	Data	3
11	[[959.215, -360.523], [959.215, -360.523], [99...	Data	3
12	[[946.139, -364.882], [946.139, -364.882], [97...	Data	3
13	[[1051.99, -326.276], [1051.99, -326.276], [10...	Text	4
14	[[1095.58, -314.446], [1095.58, -314.446], [11...	Text	4

Figure 9. First lines of a Pandas dataframe containing traces generated by `parse_inkml.py`.

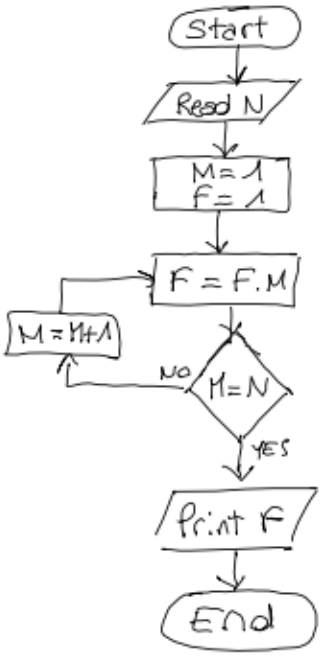

Figure 10. Plot of a InkML dataset file generated by the `plot_inkml.py` script.

3.5. Bounding boxes generation

After having converted the whole OHFCD dataset to PNG images with the auxiliary script `inkml2png.py` (which basically calls `plot_inkml.py` on every InkML file and saves the resulting plots), all that was left to do was to calculate the bounding boxes for each figure, or trace group, in the diagrams.

The `bounding_boxes.py` script [6], begins by parsing an InkML file to get all traces (as described in 3.3), then calculates for each figure its bounding boxes in *data coordinates*

[6] This Python script has become obsolete after the creation of its alternative `bounding_boxes_cmd.py` (see section 3.6.1), even though in principle they work in the same way.

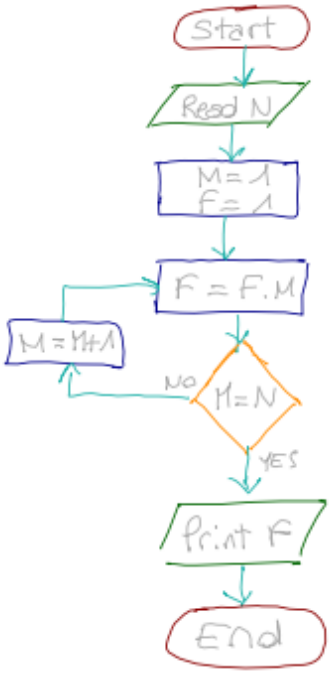

Figure 11. Coloured version of the plot in fig. 10.

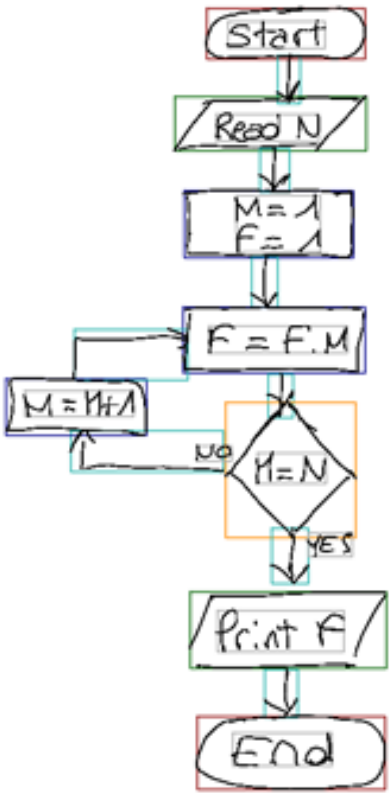

Figure 12. Bounding boxes drawn from InkML data coordinates.

by getting the maximum and minimum (x, y) coordinates of each trace group (fig. 12).

After this operation the script calls the `transform_coord.py` function, which takes the dataframe returned from `parse_inkml.py` and the bounding boxes thus calculated to return the corresponding *pixel coordinates* - the ones actually needed to work with the PNG images previously generated - by transforming data coordinates using matplotlib's `transData.transform()` [5] function. This step turned out to be quite complex, mainly due to some issues raised by the matplotlib library discussed in section 3.6.2.

3.6. Annotations and file separation

The pixel coordinates of the bounding boxes have been saved in the `annotation.csv` file (fig. 14) in the CSV format commonly used for bounding boxes, having six columns for each bounding box containing, in order: filename, minimum x coordinate, minimum y coordinate, maximum x coordinate, maximum y coordinate and the bounding box label.

This data has subsequently been separated in two CSVs, `annotation_train.csv` and `annotation_test.csv`, for the purposes of having two separate annotation files used to train and test Faster R-CNN, with the aid of the ad hoc `divide_annotations.py` script; the same has been done for all PNG images with the `divide_dataset.py` script. Both operations used the train and test list of files provided with the original InkML dataset (`listinkml_Train.txt` and `listinkml_Test.txt`).

3.6.1 Code efficiency

It has been noted that calling the `bounding_boxes()` function in a Python loop over more than 10 files at once[7] caused it to become really slow[8], so for the conversion of the entire dataset it seemed that calling it manually on 5 files at a time would have been better – but this would have been a cumbersome operation even for really patient people.

To solve this issue, an alternative bounding boxes script, `bounding_boxes_cmd.py`, has been made in order to be called from a Windows batch program (`bounding_boxes_cmd.bat`) separately on every file in a given folder.

Multi-threading might have also been an option, but it has not been tried as the previously described solution worked just fine and yielded a fast execution.

[7] As the now obsolete script `png2bb.py` used to do.
[8] Less than half dataset converted in 20 hours.

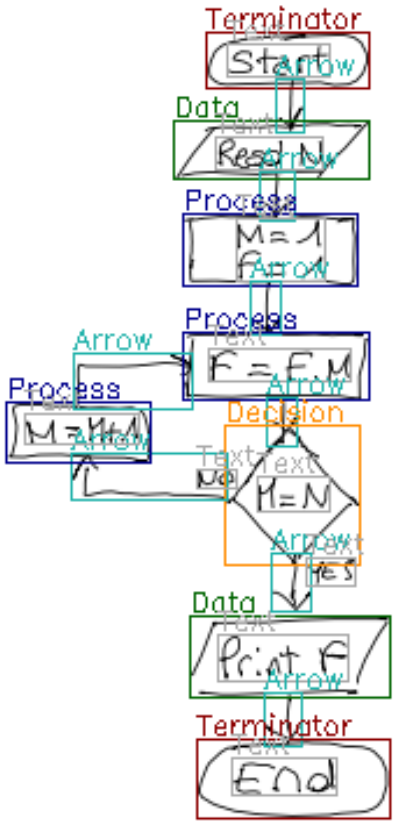

Figure 13. Bounding boxes drawn over the same diagram as figure 12, but using pixel coordinates.

```
writer1_1.png,331,73,401,97,Terminator
writer1_1.png,340,78,384,92,Text
writer1_1.png,361,93,373,116,Arrow
writer1_1.png,317,111,401,136,Data
writer1_1.png,335,118,381,132,Text
writer1_1.png,354,133,369,154,Arrow
```

Figure 14. `annotation.csv` sample.

3.6.2 About matplotlib

As mentioned before, the use of the matplotlib[7] library has caused many issues that required a great amount of work to circumvent.

First of all, since the `pyplot`[4] module of matplotlib does not allow the user to return a plot object, the `transform_coord().py` script needs to parse the InkML files and plot the flowcharts from the beginning in order to work on the resulting plots, thus producing a necessary yet unwanted redundancy[9].

Subsequently we found that using the matplotlib `transData.transform()`[5] function in a loop will cause it to return the same results even when applied to different inputs. This led to its wrapping in the `transform_coord().py` separate script, which was initially supposed to only consist of a few lines of code in its calling function, `bounding_boxes.py`.

Extensive research and experimenting have been done to attempt to circumvent this particular issue, such as implementing multi-threading to call `transData.transform()` in parallel, instead of sequentially in a loop. It turned out that neither matplotlib[6] nor Pandas[10] are thread-safe, so the only way to use `transData.transform` was the one described above.

3.7. Debugging tools

In order to debug the scripts and generate figures for this article, many utilities have been written during the project's development; below are listed those scripts that have not been named elsewhere in the previous sections.

- `add_annotation_path.py`: given a CSV annotation file with only the file names specified, adds the specified path before each file name and saves it in a new CSV.
- `bb_test.py`: draws bounding boxes over the images in the specified folder using the annotations in the specified file; it's a visual check for the bounding boxes coordinates accuracy.
- `demo.py`: calls some of the functions in the project and shows some plots; very useful for generating figures.
- `draw_cropped_bb.py`: draws all bounding boxes over the cropped images generated by the `crop.py` script; file and folder names are hard-coded, as it only serves as a visual check to ensure that the cropped coordinates have been correctly calculated.
- `draw_bb.py`: given an InkML file, it calculates the pixel coordinates of the bounding boxes using `bounding_boxes.py` and draws them over a copy of the original PNG image. It was used as a wrapping function for `bounding_boxes.py` in the original implementation of `png2bb.py`, but since the inefficiency of the latter script has been found, `draw_bb.py` has become obsolete (in favor of `bounding_boxes_cmd.py`).

3.8. Augmentation

Since our train set contains only 248 images, we use augmentation during training to improve generalization.
The transformations applied on train set are:

[9]This operation did not actually take long to execute on the CPU it was run on (an Intel Core i7-6700K @4.00Ghz), but it is worth nothing its redundancy.

- Remove text elements in flowchart images and the corresponding bounding boxes
- Randomly remove 30% of flowchart elements in train set images and their corresponding bounding boxes
- White space cropping
- Apply the following augmenters to all images in training set:
  - Add contrast
  - Horizontal flip
  - Apply affine transformations:
    - Translate by +10 relative to height/width (per axis)
    - Scale to 80% of image height/width (each axis independently)
- All of the following augmentations are executed in random order:
  - Horizontally flip 50% of all images
  - Vertically flip 20% of all images
  - Crop images by 0-10% of their height/width
  - Apply affine transformations to some of the images:
    - Scale to 80
    - Translate by +10 relative to height/width (per axis)
    - Rotate by -45 to +45 degrees

Regarding the removal of text from images and the corresponding bounding boxes, Figure15, firstly we check if InkML file belongs to train or test set, if the file belongs to test set we do not apply data augmentation.
All InkML files in train set are parsed, then for each trace we check if their class is equal to the string "Text". If it is true we do not plot it, viceversa, if it is false we plot the trace. After we check all traces we plot them and finally save the resulting images.
Bounding boxes are computed as explained above, in addiction, we check if the class of each trace is equal to the string "Text". If it is true we do not calculate the bounding boxes.

Similar to the previous data augmentation technique, we implement *"randomly remove 30% of elements in train set images"*. For each InkML file we check if it belongs to train or test set and if the file belongs to test set we do not apply data augmentation. We parse a InkML file then, we randomly remove the 30% over all elements in a InkML file. Finally, we plot it, save it and compute the bounding boxes.

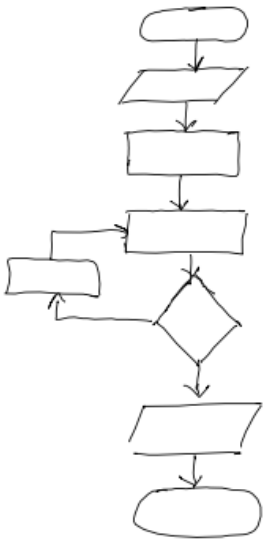

Figure 15. Data augmentation: remove text elements.

Then we implement white space cropping. The PNG images generated from the matplotlib plots contain considerable white margins around the original diagrams, so as a form of data augmentation this padding has been removed.

The script `crop_reduced.py` crops the original images by choosing, for each of them, a random cut to be applied from a total of thirteen available (one, two or three margins in all possible combinations, see figure 17). Said crop removes all white space up until the first black pixel, save for a tolerance margin of one or two pixels, and saves to `annotation_cropped_reduced.csv` the resulting bounding boxes coordinates after subtracting the number of removed pixels from the original coordinates (loaded from `annotation.csv`).

An extended version, `crop.py`, can be used to apply all thirteen cropping combinations to all images[10].

The data augmentation previously described, apart from the custom Python scripts mentioned, uses the Pandas and matplotlib libraries. Instead, the remaining data augmentation [11] uses imgaug, a library that provides features for data augmentation.

[10]See the `FCinkML_png_cropped` folder and the `annotation_cropped.csv` file in the repository.

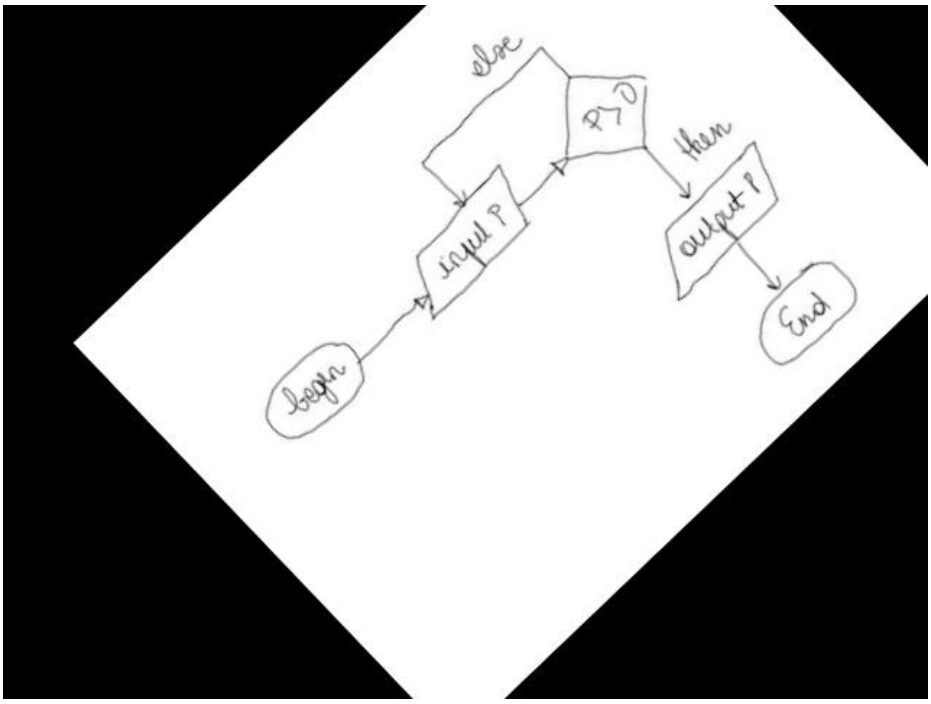

Figure 16. Data augmentation: flip, scale, translate and rotate.

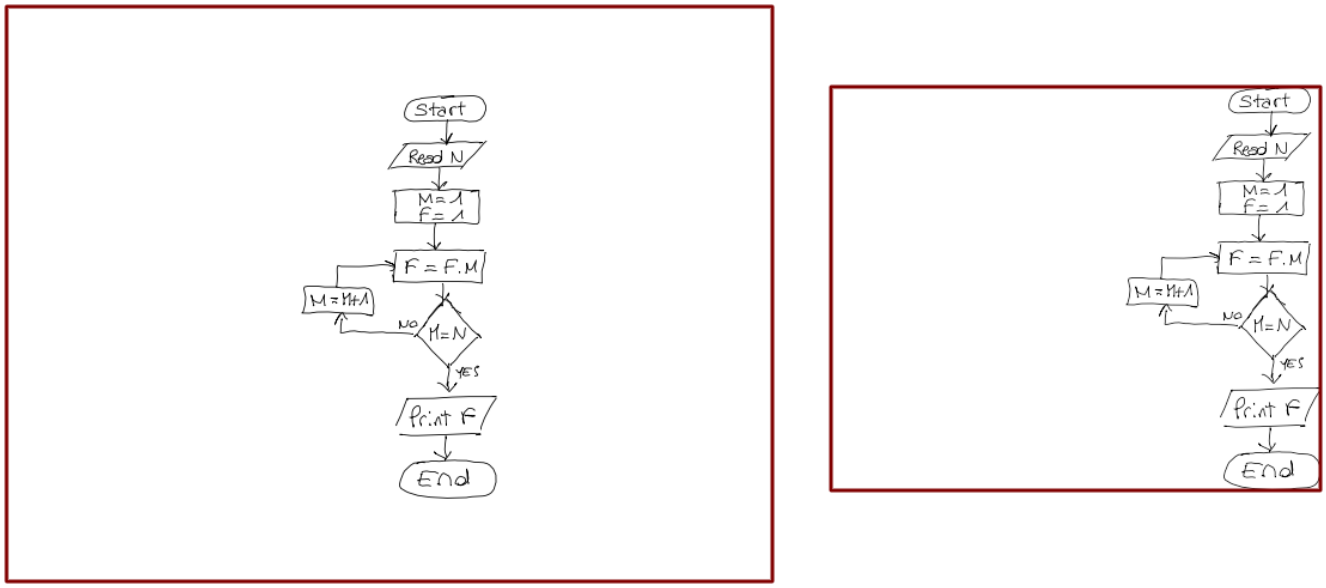

Figure 17. Example of data augmentation: original diagram (top) and its cropped version (bottom).

4. Results

4.1. First Experiment

Faster R-CNN was trained on a train set size of 248 without data augmentation and then on a train set size of 496 whose set also included augmented and not augmented images. Our network is evaluated on test set, with a size of 171 flowcharts images.

As evaluation metric we used Mean Average Precision (mAP), that is a popular metric in measuring the accuracy of object detectors like Faster R-CNN.

We performed several experiments on Faster R-CNN; re-

sults are summarized in Table 1.

As can be seen from Table 1, we evaluated Faster R-CNN on the Online Handwritten Flowchart Dataset (OHFCD) without any data augmentation on train set. Total time for train the model was around 10 hours, also due to the continuous server disconnections caused by Colab. During this experiment the network achieved a mAP of 0.597 on test set, without the aid of Data Augmentation.

Then we evaluated Faster R-CNN on the Online Handwritten Flowchart Dataset (OHFCD) using several data augmentation.

First data augmentation applied was *"remove text from images and the corresponding bounding boxes"*. Looking again at Table 1, our network achieved a mAP of 0.485 on test set. We observed that the network started overfitting early. We suppose that this is caused by the type of data augmentation applied. In particular, the augmented images have been modified by removing the "Text" element and the corresponding bounding boxes, but other elements in an image and their bounding boxes remained the same.

Another data augmentation applied was *"randomly remove 30% of elements in train set images"*. In this case, we could observe that network achieved better performance with respect to the previous data augmentation, but looking at the performance without any data augmentation, it was worse.

Subsequently we applied "*white space cropping*". The mAP is shown in Table 1. We suppose that network performance could be affected by space invariant of convolution, so crop could not add important information.

Then we evaluated Faster R-CNN on OHFCD applying the fourth set of augmenters described in the previous chapter. Faster R-CNN achieved a mAP of 0.616 on test set. We could observe how that set brings our network to perform better with respect to performance obtained with the other data augmentation.

Finally, we evaluated Faster R-CNN on OHFCD applying the fifth set of augmenters described in the previous section. An example is shown in Figure16. Faster R-CNN achieved a mAP of 0.680 on test set. We could think that, in this case, the data augmentation used has generated images that allowed our network to generalize better.

Total time for train Faster R-CNN was around 5 hours per data augmentation, also taking into account server disconnections.

4.2. Second Experiment

In the second part of our work, we decided to evaluate Faster R-CNN on a different test set. This time, test set included only flowchart images that did not contain text elements. The new test set contained 25 flowchart images. As before, in this experiment, as evaluation metric we used Mean Average Precision (mAP). Results are shown in Table 2.

	mAP
No data augmentation	0.597
Remove text	0.485
Random remove elements	0.540
White space crop	0.492
DA Set1	0.616
DA Set2	0.680

Table 1. Comparison of mAP computed on test set of Faster R-CNN. *No data augmentation*: Data Augmentation is not employed. *DA Set1*: Data augmentation set consisting of: contrast, horizontal flip, affine transformations: translation, scale. *DA Set2*: Data augmentation set consisting of: horizontally flip, vertically flip, crop images, affine transformations: scale; translate, rotate. *mAP*: Mean Average Precision.

As can be seen from Table 2, firstly we evaluated our network on the Online Handwritten Flowchart Dataset (OHFCD) without any data augmentation on the train set. Our network achieved a mAP of 0.760 on test set.

Then we applied several data augmentation techniques on the train set.

We employed *"remove text from images and the corresponding bounding boxes"* as data augmentation on train set and we evaluated our model on the new test set, described above. Our network achieved a mAP of 0.494. On a sample analysis, we observed that our network finds text elements on test set, even if it was not present.

Next data augmentation applied was *"randomly remove 30% of elements in train set images"*. In this case, we could observe that network achieved 0.591.

Subsequently we applied *"White space crop"*. The result is shown in Table 2.

Then we employed on OHFCD dataset the fourth set of augmenters described in the previous section. Faster R-CNN achieved a mAP of 0.736 on new test set. We could observe how that data augmentation allowed our network to perform better with respect to the previous data augmentation, but the performance, in this case, was lower than the performance obtained without data augmentation.

Finally, we evaluated Faster R-CNN on OHFCD applying the fifth set of augmenters described in the previous section. Faster R-CNN achieved a mAP of 0.548 on new test set.

5. Conclusion

In the first stages of this project we managed to create, by means of a series of Python scripts, a system capable of converting SVG drawings of flowcharts generated by the Neo Smartpen M1 device to InkML files that can be easily parsed and plotted with matplotlib to get, in turn, PNG

	mAP
No data augmentation	0.760
Remove text	0.494
Random remove elements	0.591
White space crop	0.643
DA Set1	0.736
DA Set2	0.548

Table 2. Comparison of mAP computed on test set composed by flowchart images with no text element, on Faster R-CNN. *No data augmentation*: Data Augmentation is not employed. *DA Set1*: Data augmentation set consisting of: contrast, horizontal flip, affine transformations: translation, scale. *DA Set2*: Data augmentation set consisting of: horizontally flip, vertically flip, crop images, affine transformations: scale; translate, rotate. *mAP*: Mean Average Precision.

images. These are subsequently given as input to Faster R-CNN, which is able to recognize flowchart elements in the diagram.

During our work, we studied Faster R-CNN, as a hand-written flowchart recognizer that is able to correctly recognize the majority of flowcharts elements in a public dataset.

Our work shows how the network performs according to train set images and data augmentation applied.

First, we showed how Faster R-CNN performs without any data augmentation methods then we applied different data augmentation to improve the performance of our network. Our experiments show that data augmentation improves the performance of the network.

Faster R-CNN achieves better performance with the fifth set of augmenters described in section 3.8 and worse with "*remove text and the corresponding bounding boxes*" described above.

In the second part of our work, we evaluated Faster R-CNN on a different test set. The new test set contained only flowcharts images without text elements. First, we have shown how Faster R-CNN performs without any data augmentation methods, then we applied the same data augmentation as before.

We could observe that Faster R-CNN on the new test set, achieved better results when we did not employ any data augmentation.

6. Appendix

In order to differentiate the contribution that each author has made to this project, a list of sections and subsections sorted by their writer is provided below. It is implied that the work described in each section has been mainly done by the relative creator, with minor suggestions from the other.

Revision of the whole work has been done by both authors, as well as all major decisions during the project's development.

The following sections have been written by Lavinia De Divitiis:

0. Abstract

2. Faster R-CNN

   2.1 Background

   2.2 Network Architecture

   2.3. Training and implementation specifications

3. Dataset

   3.8. Augmentation *(except* `crop_reduced.py` *script part)*

4. Results

   4.1 First Experiment

   4.2 Second Experiment

5. Conclusion

The following sections have been written by Paula Mihalcea:

1. Introduction

3. Dataset

   3.1. InkML file format

   3.2. SVG to InkML conversion

   3.3. InkML parsing

   3.4. InkML visualization

   3.5. Bounding boxes generation

   3.6. Annotations and file separation

      3.6.1 Code efficiency

      3.6.2 About matplotlib

   3.7. Debugging tools

   3.8. Augmentation (`crop_reduced.py` *script part*)

6. Appendix

References

[1] Colaboratory Google. `https://research.google.com/colaboratory/faq.html`.

[2] Google Colab. `colab.research.google.com`.

[3] Keras: The Python Deep Learning library. `https://keras.io/`.

[4] Matplotlib - matplotlib.pyplot. `https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.html`.

[5] Matplotlib - Transformations. `https://matplotlib.org/3.2.1/tutorials/advanced/transforms_tutorial.html#data-coordinates`.

[6] Matplotlib How-To - Working with threads. https://matplotlib.org/3.1.0/faq/howto_faq.html#working-with-threads.

[7] Matplotlib: Python plotting. https://matplotlib.org/.

[8] Neo Smartpen M1. https://www.neosmartpen.com/en/neosmartpen-m1/.

[9] Pandas - DataFrame. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html.

[10] Pandas FAQ - Thread-safety. https://pandas.pydata.org/pandas-docs/stable/user_guide/gotchas.html#thread-safety.

[11] L. De Divitiis. Data Augmentation using imgaug. https://github.com/laviniadd/Progetto_DataMining, 2020.

[12] L. De Divitiis. Faster R-CNN. https://github.com/laviniadd/Progetto_DataMining, 2020.

[13] Awal et al. First Experiments on a new Online Handwritten Flowchart Database. https://www.researchgate.net/publication/221253740_First_Experiments_on_a_new_Online_Handwritten_Flowchart_Database, 2011.

[14] Ross Girshick. Fast r-cnn, 2015.

[15] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation, 2013.

[16] P. Mihalcea. Flowcharts. https://github.com/PaulaMihalcea/Flowcharts, 2020.

[17] R. Zanibbi R. Pospesel, K. Hart. CROHME InkML Viewer (v3). http://saskatoon.cs.rit.edu/inkml_viewer/, 2019.

[18] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks, 2015.

[19] Stuckenschmidt Schäfer. Arrow R-CNN for Flowchart Recognition. https://www.researchgate.net/publication/336552832_Arrow_R-CNN_for_Flowchart_Recognition, 2019.

[20] W3C. Ink Markup Language (InkML). https://www.w3.org/2002/mmi/ink, 2011.