UI element detection from wireframe drawings of websites

Prasang Gupta[1], Vishakha Bansal[1]

[1]*PwC US Advisory, BG House, Lake Boulevard Road, Hiranandani Gardens, Powai, Mumbai, India*

Abstract

User Interfaces (UIs) wireframe is a crucial part of designing front-end of websites and mobile applications. Detection of UI elements such as paragraphs, buttons, images etc. from the wireframes using advanced Artificial Intelligence (AI) algorithms pave the way to automate the process of conversion of wireframes to Hypertext Mark-up Language (HTML) code. In this paper, we have explored different variants of 5th generation of You Only Look Once (YOLOv5) algorithm and post-processing techniques involving tuning of confidence cut-off variable for detection of UI elements. Our final approach comprises of data pre-processing using contrast normalization and conversion to black and white (BW), detection and localization of UI elements using YOLOv5x variant followed by confidence cutoff for selecting final bounding boxes. This approach resulted in Mean Average Precision (mAP) of 0.836 on the test data.

Keywords

Website UI elements, UI element extraction, Image Processing, OpenCV, Object Detection, YOLOv5, Confidence Cutoff Variation

1. Introduction

In recent times, building an online presence through websites and mobile applications has become a necessity for businesses to create global outreach and provide better customer service. Designing such applications is a time-consuming and iterative process. Wireframing serves as a starting point for this process. There are various tools that could be used for creating such wireframes and converting them automatically to code. However, the tools could be expensive and require learning for specific usage.

The wireframe task from ImageCLEFdrawnUI 2021 Task[1] which is part of ImageCLEF 2021[2] is the second edition in this area and aims at reducing the dependency on the tools and automating the code conversion process by using machine learning for detection and localization of UI elements in the wireframes. The dataset provided as part of this task has been enhanced from its previous edition in terms of volume and class distribution.

In this study, we focus on creating model-driven approach which is able to identify and localize the bounding boxes of all UI elements present in a wireframe. In the next section, we

CLEF 2021 – Conference and Labs of the Evaluation Forum, September 21–24, 2021, Bucharest, Romania

prasang.gupta@pwc.com (P. Gupta); vishakha.bansal@pwc.com (V. Bansal)
https://prasang-gupta.github.io/ (P. Gupta)
0000-0002-3623-8227 (P. Gupta); 0000-0002-5582-9145 (V. Bansal)

briefly describe the dataset used for training and validation of the models. In Section 3, we cover the methodology used âĂŞ data pre-processing, modelling and post-processing. In Section 4, we present the results from the final approach. The paper finishes with the conclusion and future work.

2. Dataset

The dataset for the ImageCLEFdrawnUI 2021 competition [1] included snapshots of hand drawn wireframe images of website layouts. These images included a total of 21 classes of atomic UI elements including images, paragraphs, headers, links etc. The provided dataset included 4291 of such images. These images were divided into a development and a test set.

Table 1
Class distribution in the train and validation sets.

Label	Train Freq	Val Freq	Train Dist	Val Dist
paragraph	2727	141	0.89	0.88
label	1004	42	0.33	0.26
header	2059	121	0.67	0.76
button	2991	153	0.98	0.96
image	2462	121	0.81	0.76
linebreak	2370	128	0.78	0.8
container	2923	153	0.96	0.96
link	1031	56	0.34	0.35
textinput	1453	69	0.48	0.43
dropdown	688	34	0.22	0.21
checkbox	663	25	0.22	0.16
radiobutton	478	14	0.16	0.09
rating	434	11	0.14	0.07
toggle	452	13	0.15	0.08
textarea	418	9	0.14	0.06
datepicker	468	12	0.15	0.08
stepperinput	91	3	0.03	0.02
slider	491	16	0.16	0.1
video	448	22	0.15	0.14
table	56	1	0.02	0.01
list	180	6	0.06	0.04

The development set contained 3218 labelled images while the test set contained 1073 unlabelled images. The development set was further divided into a train and a validation set. Out of the 3218 images in the development set, 3058 images were included in the train set and the rest 160 images formed the validation set (about 5% of the development set). This division was done keeping in mind that the distributions of the classes remain as close as possible. The exact distribution is shown in Table 1.

The images were all RGB images having a myriad of different sizes. The size distribution of

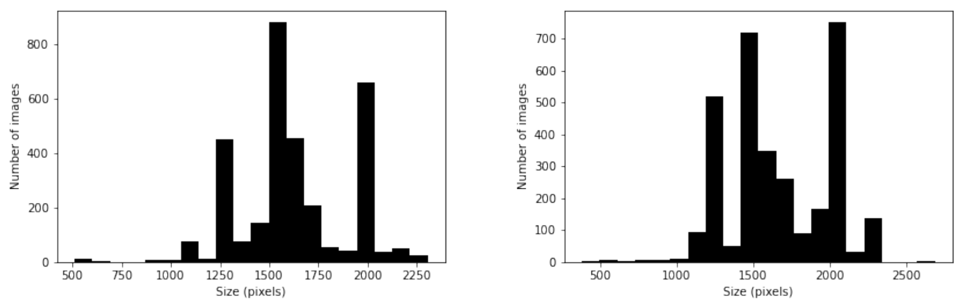

Figure 1: Distribution of heights and widths of the images in the development set with 20 bins.

the images can be seen in the histograms for height and width distribution in Figure 1. All the images were later resized to a constant size of 512 x 512 for training purposes.

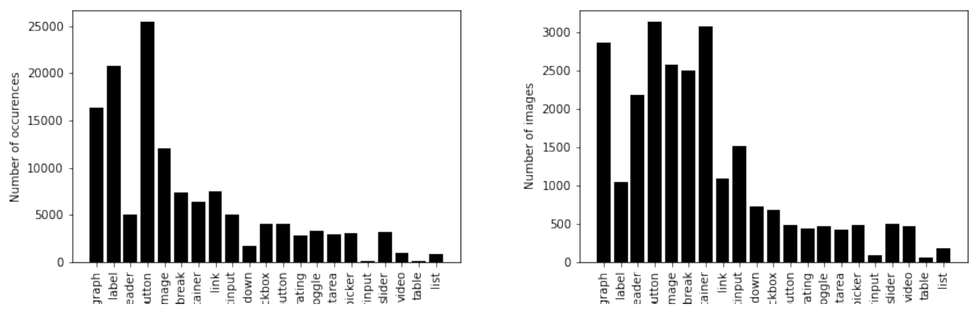

Figure 2: Distribution of the classes within the development dataset. The plot on the left shows the total number of occurrences of a class in the dataset. The plot on the right shows the spread of the classes defined as the total number of unique images which have atleast 1 occurence of that class. Both of these plots show that some classes are abundantly present while some are a little less represented.

The 21 classes present had different amount of representation in the dataset. Some classes which are commonly found in websites were abundantly present and dominated most of the images in the dataset as well. These classes are paragraph, button, link, image etc. However, some classes which are not as abundantly present in websites such as table, video, stepperinput, list etc. were present in comparatively lesser number in the dataset as well. The distribution of the classes among the dataset can be seen in Figure 2.

3. Methodology

3.1. Data Pre-Processing

There are different pre-processing techniques that we have employed to improve the viability and performance of our model. As we are dealing with image data, most of our pre-processing steps use OpenCV. We have opted to use OpenCV [3] on C++ because of the added speed it provides when dealing with large images. Some of the techniques we have used are described in the subsequent sections.

3.1.1. Contrast improvement

The images present in the dataset are made by taking snapshots of wireframe drawings of websites drawn by users. These drawings were either made on paper using pen or on a whiteboard using a marker. As the final images are snapshots, they very much depend on the quality of the camera used. Since most of the cameras introduce noise or a brightness overlay on the image, it was expected and was verified that different images had different tints, brightness and contrast.

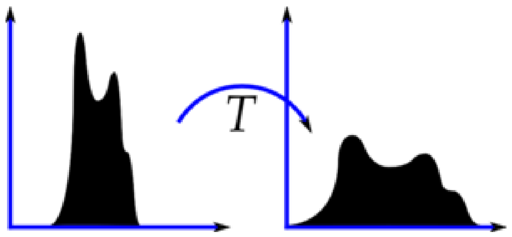

Figure 3: Histogram equalisation technique changing the narrow histogram of the pixels in the input image to a much more wide distribution.

To counter this issue, there are histogram equalisation techniques which change the range of the pixels present in the image from a very confined space to a much larger distribution. This generally results in a much clearer image with better separation. This can be visualised in Figure 3.

The histogram equalisation technique is powerful, but it also has a shortcoming. In addition to enhancing the contrast of the image, it also enhances the noise. Since our dataset has a lot of noise due to the nature of how they are collected, we chose to employ the Contrast Limited Adaptive Histogram Equalisation(CLAHE) technique from the OpenCV library which overcomes this shortcoming.

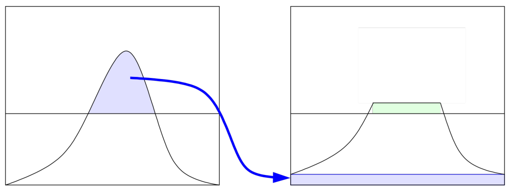

Figure 4: Histogram equalisation after contrast limiting (CLAHE)

Contrast Limited Adaptive Histogram Equalisation (CLAHE) technique is a modification of the more general histogram equalisation techniques. It employs contrast limiting before applying histogram equalisation. This is achieved by clipping the histogram bins which are

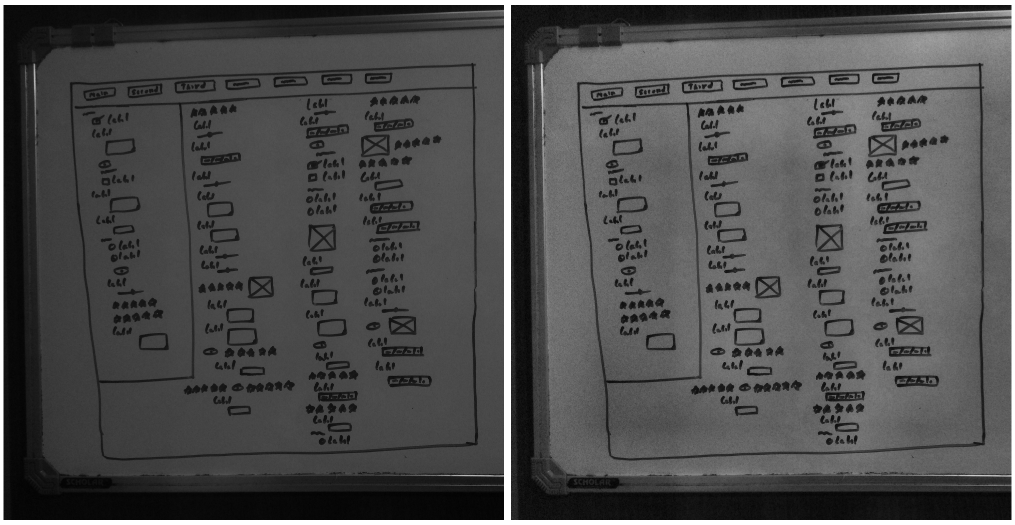
Figure 5: A depiction of the effect of CLAHE on images. The image on the left is the original image and the one on the right is after applying CLAHE. It can be observed that the processed image is much more clear and legible than the original.

above a specified contrast limit (we have used the default value 40 in this study) and then uniformly distributing the clipped pixels to the other bins. This can be visualised in Figure 4. Using this technique provided us with much more cleaner images for our dataset. The effect of this technique can be visualised on a sample image from the dataset in Figure 5.

3.1.2. Conversion to Black and White

After getting the contrast normalised images, the next step was to convert them into black and white. This would remove any noise present in the image and focus the model to identify only the things that matter i.e. the wireframe drawings.

There were a lot of iterations performed before by Gupta et. al. [4] to get the most effective way to convert wireframe grayscale images to black and white. We have directly used the techniques discussed in that paper.

3.2. Modelling

This problem is an object detection problem at its heart. Hence, a lot of object detection models like Mask-RCNN [5], YOLO [6] and EfficientDet [7] come to mind. This problem can also be modelled as a segmentation problem. Hence, this adds other famous and proven models like U-Net [8], LinkNet [9], FPN [10] and PSPNet [11] to the list of possible modelling techniques.

For the purpose of this problem, we started with exploring U-Net and Mask-RCNN. The U-Net model is proven to work well on object detection problems, but considering that the size of the dataset is not huge as compared to deep learning (DL) standards, training a full U-Net would firstly, take a lot of time and secondly, would run into overfitting issues due to the sheer number of parameters involved.

Mask-RCNN was another great option to go ahead with. It is much easier to train and has been known to perform at par, if not better than U-Net on different use cases [12] [13]. However, as it has already been explored and found to run into problems in detecting smaller UI elements, it was discarded [4].

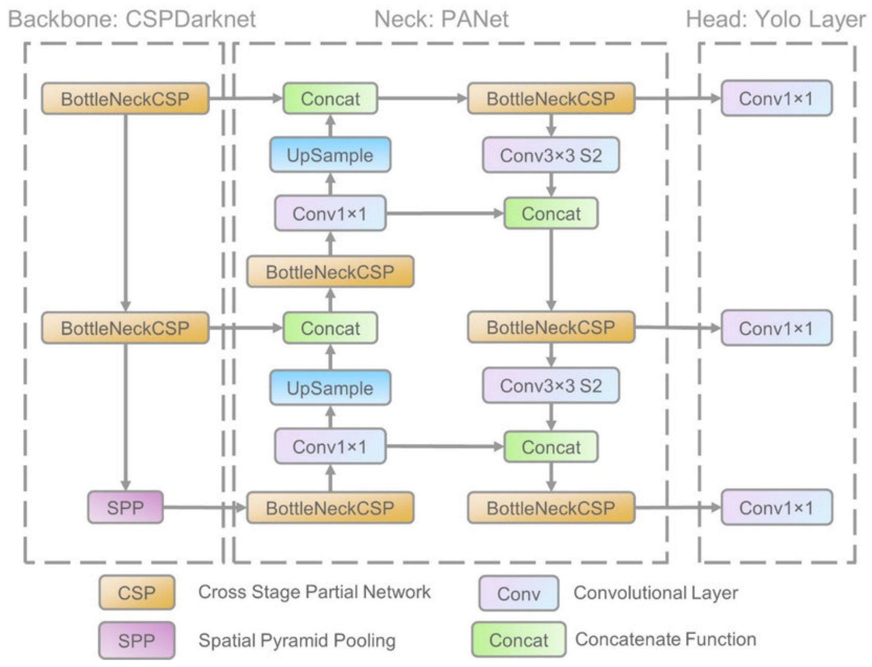

Figure 6: YOLOv5 architecture [14]

Table 2
Comparison of different variants of YOLOv5 using COCO dataset[15]

Model	mAP val 0.5:0.95	mAP Test 0.5:0.95	mAP val 0.5	Speed on V100 (ms)	Parameters (millions)
YOLOv5s	43.3	43.3	61.9	4.3	12.7
YOLOv5m	50.5	50.5	68.7	8.4	35.9
YOLOv5l	53.4	53.4	71.1	12.3	77.2
YOLOv5x	54.4	54.4	72.0	22.4	41.8

We chose to go with the latest version of YOLOv5 [15] to ensure speedy inference for real-life use cases as well as flexibility in choosing the right number of parameters based on its different flavours. The general architecture of YOLOv5 is shown in Figure 6. Also, YOLOv5 is available in 4 different size variants, the details of which are present in Table 2 most of which has been explored in this study.

Before diving into the details of all the different submissions made, let us go over the common elements of all the runs. The images were all resized to size 512 x 512 using OpenCV's linear interpolation method after performing the aforementioned pre-processing steps. Also, the dataset split was chosen to be about 95% for train and 5% for validation. A batch size of 32 was chosen throughout to train except for the extra large YOLO variants where a batch size of 16 was used. The models were trained on Google Colab using a GPU environment with a single NVIDIA Tesla K80 GPU.

3.2.1. Run 1 : YOLOv5s Baseline

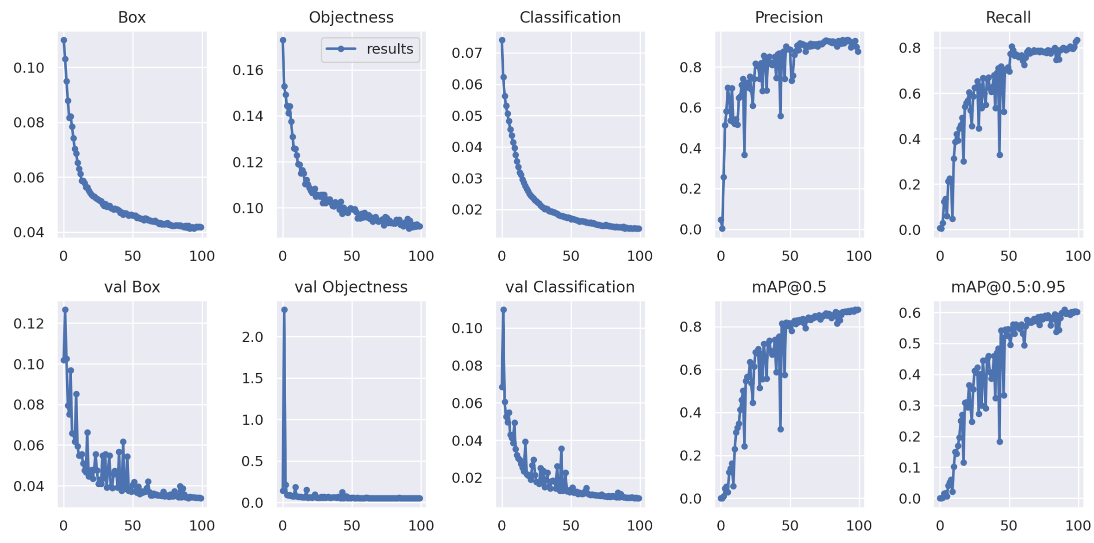

Figure 7: Train results for the YOLOv5s model with no pre-trained weights.

To establish a baseline for our runs, we used a basic YOLOv5s architecture with no pre-trained weights and trained the model for 100 epochs from scratch. The training metrics for the model can be seen in Figure 7. We got a precision score of 0.876 and a recall score of 0.835 on our validation dataset. These numbers were decent for a small model with no starting weights. To explore further on how pre-trained weights would affect this, we incorporated that in the next run.

3.2.2. Run 2 : YOLOv5s with pre-trained weights

This run included pre-trained weights on the same YOLOv5s architecture, These pre-trained weights were originally generated by training the model on COCO dataset [16]. COCO dataset is very general in nature with 91 classes and contains 123,287 images. Having been trained on such a large gamut of images, these weights have a lot of information already encoded in them. Hence, very little training suffices for a decent performing model.

We loaded the model with the pre-trained weights and trained it for 100 epochs keeping all the layers unfrozen. The training metrics for the model can be seen in Figure 8. We got a significant bump in our validation dataset metrics with a precision score of 0.951 and a recall score of 0.82 on the same.

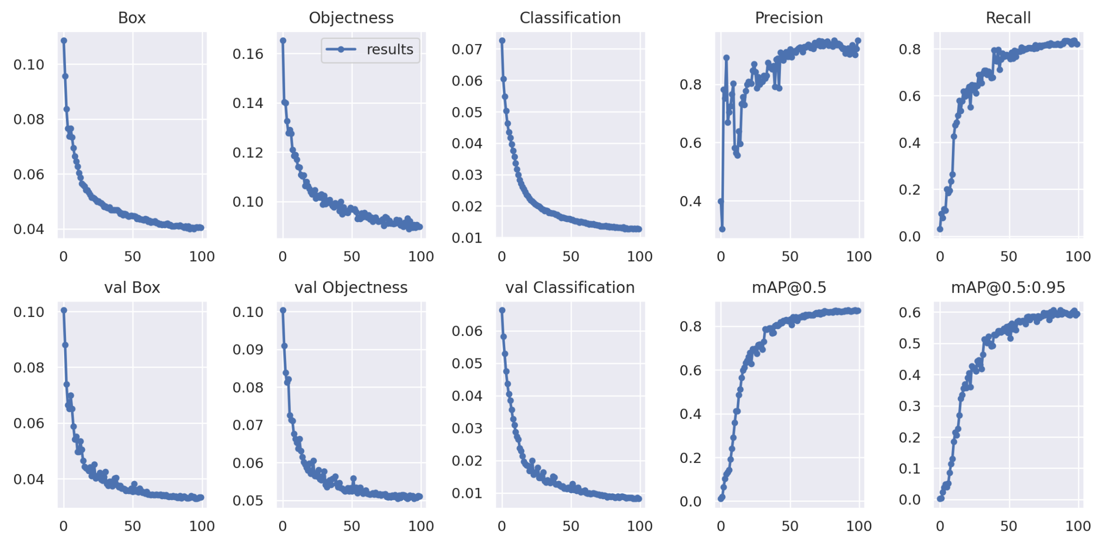

Figure 8: Train results for the YOLOv5s model with pre-trained weights.

3.2.3. Run 3 : YOLOv5l

Having tried out the small variant of YOLOv5, we moved on to the large variant. However, this time we also employed a learning rate scheduler. We used pytorch's [17] implementation of *ReduceLROnPlateau*. We also implemented early stopping in this run. Both of these settings were carried forward to the rest of the runs as well.

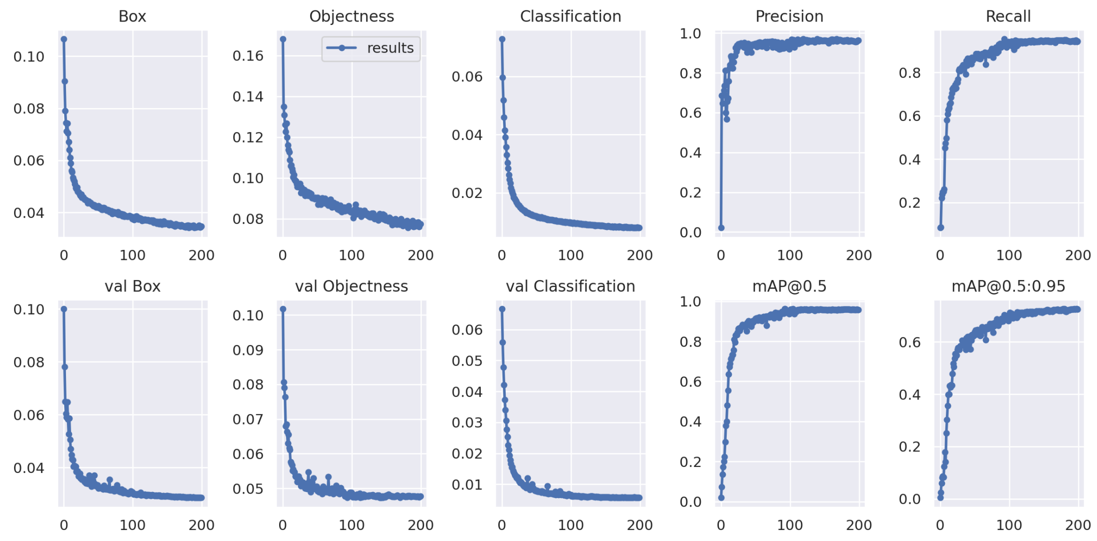

Figure 9: Train results for the YOLOv5l model with pre-trained weights.

We trained the large model with the pre-trained weights for 200 epochs. The training metrics for the model can be seen in Figure 9. We got a precision score of 0.964 and a recall score of 0.944 on our validation dataset. This is an improvement over the small model which was expected because of the larger parameters present in the large variant.

3.2.4. Run 4 : YOLOv5x

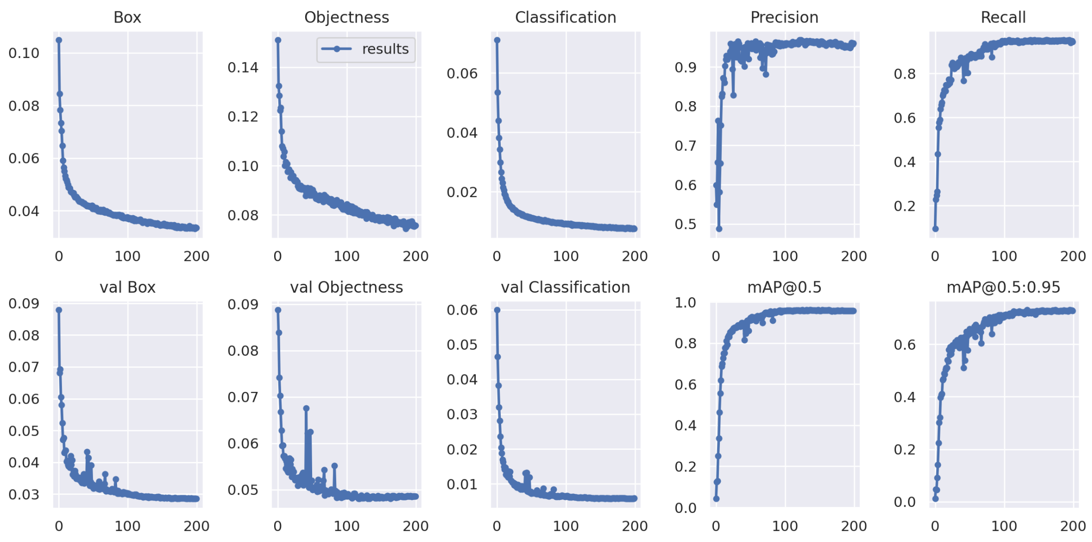

Figure 10: Train results for the YOLOv5x model with pre-trained weights.

To extract the highest amount of performance from the YOLO models, we next tried the YOLOv5x variant which is the largest variant present. Because of the huge size of the model parameters, we reduced our batch size to 16. We trained this model with pre-trained weights for 200 epochs and got very similar performance on our validation dataset. The training metrics for the model can be seen in Figure 10. We got a precision score of 0.961 and a recall score of 0.943 on the same.

However, the difference was notable in the test performance of the extra large and the large model. The large model got an mAP value of 0.81 while the extra large model got an mAP score of 0.82 on the test set. Even though there are diminishing returns with respect to the speed of the model, as speed was not an issue, we decided to go ahead with the extra large model for further investigation.

3.2.5. Run 5 : YOLOv5x with frozen layers

Up until now, we were loading the pre-trained weights into the model, and re-training all the layers. In this run, we decided to freeze the early layers and train only the head of the model. This would ideally ensure much faster training time as the number of parameters to be updated currently are huge.

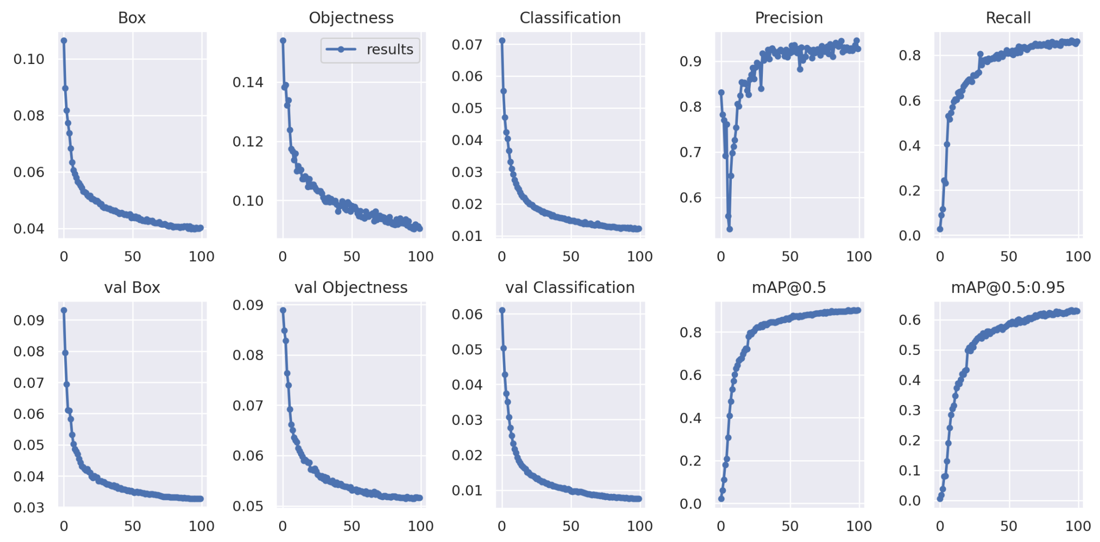

Figure 11: Train results for the YOLOv5x model with pre-trained weights and layers 0-9 frozen (Trained only on the head layer)

We trained only the head (with 0-9 layers frozen of the model) for 100 epochs. The training was much faster, however their was a dip in performance. The training metrics for the model can be seen in Figure 11. We got a precision score of 0.927 and a recall score of 0.863 on the validation dataset. Since this was a sizeable dip, we planned to go ahead with the model we got in Run 4.

3.3. Post-Processing

After getting the final trained model in Run 4, their were several post-processing methods that were employed. The first method that was employed was the multi-pass inference [4] and the second method was model confidence variation. We will discuss both of these methods in detail in the following sections.

3.3.1. Multi-Pass Inference

This method is predominantly used for increasing the recall score. This technique works by sending the image through the model multiple times, each time removing the objects that were detected earlier and then smartly appending all the outputs together based on different confidence scores and pass numbers.

We employed this technique to our study, but this did not work well as we found that our model was performing very well in the recall department detecting almost all the UI elements. Hence, there was little to no scope for improvement using this technique. Hence, we scrapped this and went on to our next post processing method.

3.3.2. Confidence cutoff variation

Another variable that was found to be important for the performance of our model was the confidence cutoff variable. This cutoff is a hyperparameter for the model. and is responsible for selecting all the bounding boxes that appear in the final results of the model while the others are discarded.

Table 3
Different cutoff values tried and corresponding number of labels and test mAP score.

Confidence Cutoff	Total Dataset labels	% increase in labels	Test mAP scores
25	101251	0.0	0.820
20	109848	8.5	0.824
15	119881	9.1	NA
10	130765	9.0	0.829
5	142963	9.3	0.832
1	165013	15.4	0.836

There were several different cutoff values we tried with different number of total labels detected in the model. We observed a marginal increase in the performance of the model. The cutoff variable was changed from 25% to 1%. The summary of the results is shown in Table 3.

4. Results and Discussion

A total of 10 submissions were made. The predictions on the test set images were collated in a csv file. For each image on the test set, the bounding boxes corresponding to each instance of a detected class and the confidence scores were submitted. The mAP and recall scores obtained on the test dataset are shown in Table 4.

Table 4
Table summarising the runs submitted for the challenge.

	Run ID	Model Description	mAP	Recall
Run 1	132552	YOLOv5s baseline	0.649	0.675
Run 2	132567	YOLOv5s with pre-trained weights	0.649	0.675
Run 3	132575	YOLOv5l with pre-trained weights	0.810	0.826
Run 4	132583	YOLOv5x with pre-trained weights , LR, Early Stopping	0.820	0.840
Run 5	132592	YOLOv5x with pre-trained weights and only heads trained	0.701	0.731
Run 6	134090	Run 4 with 0.2 confidence cutoff	0.824	0.844
Run 7	134099	Run 4 with 0.15 confidence cutoff	0.824	0.844
Run 8	134113	Run 4 with 0.1 confidence cutoff	0.829	0.852
Run 9	134133	Run 4 with 0.05 confidence cutoff	0.832	0.858
Run 10	134133	Run 4 with 0.01 confidence cutoff	0.836	0.865

It has be seen that YOLOv5x performed best of all the YOLOv5 variants and confidence cutoff variable used for post processing is an important factor as it contributed to an increase in the performance of the model.

5. Conclusion

In this paper, we have built YOLOv5 based model to detect and localize UI elements in wireframes. We also performed contrast normalization for improvement in the quality of the input images to the model and introduced tuning of confidence cut-off variable for improving the output performance of the model. An mAP score north of 0.8 was attained using This approach on the test data which consisted of wireframes containing a range of UI elements helping us gain 2nd position on the leaderboard of wireframe task of ImageCLEFdrawnUI 2021 [1].

This approach could be integrated further into the pipeline of automating the conversion to front-end code and ensure speedy inference for real-life use cases. There is also a scope of experimenting with the ensemble of two modelling approaches: one for wireframes with more compactly placed UI elements and another for wireframes with less compactly placed UI elements. This would ensure that confidence cutoff variable is correctly tuned and would result in getting the reasonable number of selected bounding boxes for these two different cases.

Acknowledgments

Thanks to the developers of Google Colab for providing a free GPU environment for model training.

References

[1] R. Berari, A. Tauteanu, D. Fichou, P. Brie, M. Dogariu, L. D. Ştefan, M. G. Constantin, B. Ionescu, Overview of ImageCLEFdrawnUI 2021: The detection and recognition of hand drawn and digital website uis task, in: CLEF2021 Working Notes, CEUR Workshop Proceedings, CEUR-WS.org <http://ceur-ws.org>, Bucharest, Romania, 2021.

[2] B. Ionescu, H. Müller, R. Péteri, A. Ben Abacha, M. Sarrouti, D. Demner-Fushman, S. A. Hasan, S. Kozlovski, V. Liauchuk, Y. Dicente, V. Kovalev, O. Pelka, A. G. S. de Herrera, J. Jacutprakart, C. M. Friedrich, R. Berari, A. Tauteanu, D. Fichou, P. Brie, M. Dogariu, L. D. Ştefan, M. G. Constantin, J. Chamberlain, A. Campello, A. Clark, T. A. Oliver, H. Moustahfid, A. Popescu, J. Deshayes-Chossart, Overview of the ImageCLEF 2021: Multimedia retrieval in medical, nature, internet and social media applications, in: Experimental IR Meets Multilinguality, Multimodality, and Interaction, Proceedings of the 12th International Conference of the CLEF Association (CLEF 2021), LNCS Lecture Notes in Computer Science, Springer, Bucharest, Romania, 2021.

[3] G. Bradski, The OpenCV Library, Dr. Dobb's Journal of Software Tools (2000).

[4] P. Gupta, S. Mohapatra, Html atomic ui elements extraction from hand-drawn website images using mask-rcnn and novel multi-pass inference technique (2020).

[5] K. He, G. Gkioxari, P. DollÃąr, R. Girshick, Mask r-cnn, in: 2017 IEEE International

Conference on Computer Vision (ICCV), 2017, pp. 2980–2988. doi:10.1109/ICCV.2017.322.

[6] J. Redmon, S. Divvala, R. Girshick, A. Farhadi, You only look once: Unified, real-time object detection, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 779–788. doi:10.1109/CVPR.2016.91.

[7] M. Tan, R. Pang, Q. V. Le, Efficientdet: Scalable and efficient object detection, 2020. arXiv:1911.09070.

[8] O. Ronneberger, P. Fischer, T. Brox, U-net: Convolutional networks for biomedical image segmentation, in: N. Navab, J. Hornegger, W. M. Wells, A. F. Frangi (Eds.), Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015, Springer International Publishing, Cham, 2015, pp. 234–241.

[9] A. Chaurasia, E. Culurciello, Linknet: Exploiting encoder representations for efficient semantic segmentation, 2017 IEEE Visual Communications and Image Processing (VCIP) (2017). URL: http://dx.doi.org/10.1109/VCIP.2017.8305148. doi:10.1109/vcip.2017.8305148.

[10] T.-Y. Lin, P. DollÃąr, R. Girshick, K. He, B. Hariharan, S. Belongie, Feature pyramid networks for object detection, in: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 936–944. doi:10.1109/CVPR.2017.106.

[11] H. Zhao, J. Shi, X. Qi, X. Wang, J. Jia, Pyramid scene parsing network, in: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 6230–6239. doi:10.1109/CVPR.2017.660.

[12] M. S. Durkee, R. Abraham, J. Ai, J. D. Fuhrman, M. R. Clark, M. L. Giger, Comparing Mask R-CNN and U-Net architectures for robust automatic segmentation of immune cells in immunofluorescence images of Lupus Nephritis biopsies, in: I. Georgakoudi, A. Tarnok (Eds.), Imaging, Manipulation, and Analysis of Biomolecules, Cells, and Tissues XIX, volume 11647, International Society for Optics and Photonics, SPIE, 2021, pp. 109 – 115. URL: https://doi.org/10.1117/12.2577785. doi:10.1117/12.2577785.

[13] T. T. P. Quoc, T. T. Linh, T. N. T. Minh, Comparing u-net convolutional network with mask r-cnn in agricultural area segmentation on satellite images, in: 2020 7th NAFOSTED Conference on Information and Computer Science (NICS), 2020, pp. 124–129. doi:10.1109/NICS51282.2020.9335856.

[14] R. Xu, H. Lin, K. Lu, L. Cao, Y. Liu, A forest fire detection system based on ensemble learning, Forests 12 (2021) 217. doi:10.3390/f12020217.

[15] G. Jocher, A. Stoken, J. Borovec, NanoCode012, A. Chaurasia, TaoXie, L. Changyu, A. V, Laughing, tkianai, yxNONG, A. Hogan, lorenzomammana, AlexWang1900, J. Hajek, L. Diaconu, Marc, Y. Kwon, oleg, wanghaoyang0106, Y. Defretin, A. Lohia, ml5ah, B. Milanko, B. Fineran, D. Khromov, D. Yiwei, Doug, Durgesh, F. Ingham, ultralytics/yolov5: v5.0 - YOLOv5-P6 1280 models, AWS, Supervise.ly and YouTube integrations, 2021. URL: https://doi.org/10.5281/zenodo.4679653. doi:10.5281/zenodo.4679653.

[16] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, P. DollÃąr, Microsoft coco: Common objects in context, 2015. arXiv:1405.0312.

[17] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, Pytorch: An imperative style,

high-performance deep learning library, in: H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, R. Garnett (Eds.), Advances in Neural Information Processing Systems 32, Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.