Synthetic Datasets for Instance Segmentation

ComputerVision

Diaconu Andrei
University of Trento - M.Sc. Computer Science
andrei.diaconu@studenti.unitn.it

Iossa Andrea
University of Trento - M.Sc. Computer Science
andrea.iossa@studenti.unitn.it

***Abstract*—In this paper we describe the process of generating ground truth images of persons for segmentation tasks in a simulated environment using the Unity game-engine. We present our results of the training of a mask R-cnn model using the Detectron2 library on our generated datasets and how those compare to baseline models trained exclusively on real images from the COCO dataset.**

***Index Terms*—COCO, Detectron2, mask R-cnn, instance segmentation, simulation, Unity**

I. Introduction

In the context of image segmentation with deep learning, like with other tasks related to computer vision, the emphasis is generally put on the new models that are constantly being developed and published, but another aspect often undervalued is how these models are trained or, better on what are they trained. To train a segmentation algorithm on a dataset of training images, more complex the model more images are needed. The task of collecting different images is hard on his own, to train a model effectively you need images of different complexity with lots of subjects of different entities, but to make things even more complicated, for each image it is necessary to provide the segmentation information of each object. There are no automated processes reliable enough to deal with complex images such us crowds of people or lots of different classes of subjects, so the only real way is to proceed manually. If we consider for example the COCO dataset, one of the most widely used dataset in these kind of contexts for its incredibly high number of images and object instances, there are more than 200.000 images and more than a staggering 1.5 million objects that needed to be segmented manually. By looking to these numbers is easy to understand how complex and challenging the task of building such a dataset is and how many resources are needed to succeed.

Datasets such as COCO are incredibly useful for a lot of researches because are free to download and use but sometimes there is the necessity to build a custom dataset that can satisfy more niche needs. What we are trying to see is if these processes of collecting images and manually label them can be substituted with a more easy and fast approach, simulation. Using modern 3D engines such us Unity, is possible to model whatever object is deemed necessary, construct scenes of varying complexity and most importantly it provides an easy way to create a ground truth image and label the objects in it.

A. The scope of our project

What we aim to study with this research is how viable synthetic images are in the context of person segmentation. We produced a dataset of generated images of small crowds of Unity Multipurpose Avatars (UMA[1]) using a person simulator developed on the Unity game engine, expanding it with a new component build specifically for segmentation tasks. We then tried to use these images to both train a Mask R-cnn segmentation model, specifically using the Detectron2[2] library from facebook, from scratch and for fine-tuning the already trained model from the same library. A positive outcome of these tests would mean that simulated images are a viable alternative to manually collected datasets, effectively easing the training of other algorithms of this kind.

To help others expand our research, in this paper we also provide the information necessary to understand how our component on the simulator works as well as how to proceed with collecting the images, producing the annotations for the dataset and proceed with the training.

II. Images acquisition and ground truth generation

For the generation of synthetic data and ground truth, an Unity project called Crowd-Simulator[3] was used and extended according to our needs. The mentioned project simulates a dynamic animated urban environment with crowds of people that move accordingly to a social force model. It comes with a saved scene representing an urban area that would recall Trento somehow, and uses the UMA system for the generation of randomized humans. For the purposes of our project, we built upon a new branch called *static-simulation*.

A. Crowd Generator custom branch

We reused the scene of Trento and the UMA generator, but we discarded the social force simulator with its animations as its computational impact is substantial and it was not needed in our context. Our objective was to produce a dataset of image that was somewhat comparable to the real COCO dataset. For this reason we wanted to obtain a large number of images that needed to be varied enough to resemble a real dataset. We decided to spawn the avatars with randomly generated coordinates and orientation at each frame of the simulation in order to achieve variety in the number of subjects as well as size and perspective in the images. In order to further augment this similarity with the real dataset we decided to introduce to the simulator a new animation controller to assign at each

UMA object. We imported some general purpose pre-made animations assets and we linked them to the controller. At each frame of the simulation each avatar has a given chance to shift to one of the newly imported animations at random so that the images collected may look more realistic as a whole. To add some illumination variety to the images, we used the Azure Sky Manger module that came with Crowd Generator, and set up a day-night cycle.

In addition, the *"Shake Camera"* script was applied to each camera to increase the effective number of point of views. This module can be used to make random changes in the position and direction of a camera, with customizable intensities and velocities.

Finally, we did not use a customized UMA generator as we found it was too burdensome to change and obtain realistic humans from it. Outside UMA, more realistic models available online could have been used, but is hard to collect them in enough quantities to make even a modest crowd. Thus, we kept the default UMA generator and kept in consideration the fact that this was probably the element that limited the realism of our crowds the most.

B. Obtaining Segmentation Masks

For rendering and saving the segmentation masks, we had to dig a bit deeper because the related solutions we found are not compatible with URP, the universal rendering pipeline which Crowd Generator uses. To make this work, a new renderer was added to the *Universal Render Pipeline Asset*, and a URP compatible unlit material was created and assigned to that new layer. Having done so, any camera that renders in this layer, would see the scene as every object had its material overridden by the unlit one, so we made a clone for every camera used in our scene, and assigned this new renderer layer to each one.

But at this point, all the cloned cameras would see everything of the same color: in order to objatin a proper segmentation mask, after having instantiated the UMAs, an unique color is assigned to each person for the material used in renderer pass devoted to the segmentation rendering. For every other object in the scene the color black was used in such renderer.

Figure 1. Example of ground truth as seen by a camera

Algorithm 1: RLE for COCO annotations

```
input: Image
begin
    BGcolor ⟵ (0,0,0);
    colorDict ⟵ {};
    colorDict[BGcolor] ⟵ [0];
    pixelCount ⟵ 0;
    lastColor ⟵ BGcolor;
    for x ←1 to Image.width do
        for y ←1 to Image.height do
            clr ← Image[x, y];
            if lastColor != clr then
                colorDict[lastColor].append(0);
                if clr ∈ colorDict then
                    colorDict[clr].append(0);
                else
                    colorDict[clr] ← [pixelCount, 0];
            foreach element ∈ colorDict do
                element.last++;
            lastColor ← clr;
            pixelCount++;
```

The code used to screenshot the default renderer was recycled as the logic does not change for the cloned cameras, their difference is just using a different renderer layer.

The result is that now there are two render passes in the scene, one for the default shaded representation, and one used for the ground truth of the instance segmentation task, and as such, they are synchronized. An example of our ground truth image for instance segmentation can be seen in Fig.1.

C. Converting ground truth to COCO Annotations

At this point we are able to generate synthetic images and, with them, the ground truth. But the ground truth for instance segmentation is rarely used directly as an image; in fact the different frameworks for deep learning in computer vision usually require them encoded as part of the annotation file, and that is also the case of Detectron2, which has builtin support for COCO styled annotations. This file tells the trainer what objects are in each image and which pixels belong to that objects. This process is usually done by manually selecting the pixels in our simulated environment we can use specific algorithms to parse our ground truth images to obtain the annotations.

There are two way to compute the segmentation annotations: polygons, a list of vertices around the object and Run Length Encoding (RLE). RLE masks are computed considering the image as a one dimensional array of pixels and counting at which pixel the object start, how many background pixels are encountered first, this values becomes the first in the list and

Figure 2. Samples from our training dataset, with associated instance segmentation mask

at this point the number of subsequent pixels belonging to the object is counted and this value is again added to the list. The process is carried out until the last pixel of the image is checked. Is easy to understand how our ground truth images are perfect to obtain RLE masks, and the main advantage of using this method is that with a single pass we can compute the masks for each object in the scene, resulting in a faster computation of the annotations.

We developed an RLE segmentation algorithm that can be found in the *"segMaskToCOCO.py"* file, which is separated from the files of the simulator This process is in fact designed to be applied after the images acquisition giving is time consuming aspect. The algorithm, developed in python, is pretty simple in its design. The color in the ground truth image is used to recognize the different objects displayed, while keeping the color *black* as our background constant. We start from the top left corner of the image and we parse each pixel vertically column by column, keeping track of every color observed and computing the RLE masks simultaneously, storing each entry in a dictionary structure. To better understand our algorithm refer to the pseudocode in Algorithm 1.

III. Datasets

The final training set contains about 2500 images for each camera, making a total of 15k images, each having a resolution of 960x720. Fig.2 shows a random image and associated segmentation mask taken randomly for each camera of the training dataset. In addition to the training dataset we produced a synthetic test dataset to evaluate the performances of the different models on simulated persons. The testing set images have the same resolution, but they are about a couple of thousands.

A. Preparing the real dataset

In order to really assess the performances of our procedure we needed to have a dataset of real images for comparison and testing. The testing COCO dataset was reserved for the 2017 competition and the annotations for it were not provided, we then decided to use a subset of the training COCO datasets to conduct our tests. The COCO datasets contain images of 80 categories so an additional filtering was necessary. Firstly we needed to remove images that did not contain persons at all, the remaining images needed to be strip out of the annotations relative any other category. This process was handled with a python script, with the help of the COCO API, which is the fastest and easiest way to parse an annotation json file effectively. This script and the other used after it are available at *"datasets/coco/coco.py"*.

One thing we noticed our generated dataset was lacking with respect to the real one, were close-up images or portrait of individuals. In an attempt to reduce the gap between the two datasets we impose an additional filter on the real one. We chose a greedy approach and discarded all images that contained less than 5 people in it, attempting to keep only images of crowds of varying size. The final result was a dataset of 64000 images, effectively half of the original training COCO dataset. All additional real datasets we used were produced using this one as baseline.

B. The mixed dataset

We also wanted to study how our synthetic images could be used as a form of augmentation in a pre-existing real dataset. To test this we created a "mixed dataset" from a subset of our training real images and the generated ones. We wanted to evenly split the dataset so that half of the annotations were of real persons and the other half of synthetic ones. This means that the point of division was not strictly the number of images of each type but how many real and generated persons were in them comprehensively. To be able to merge the two annotation files relative to the two datasets another python script was developed, *"merge.py"*. When an instance of training is started it is necessary to input the format of the segmentation mask of the annotations, this being RLE or polygons. We already distress how RLE was easier and more convenient to compute but the standard COCO format uses polygons for the annotation. We therefore develop a set of function to convert the RLE format to a polygon one, a step necessary to proceed with the merge of the datasets. All the function used for the conversion and the merging are available in the before mentioned script.

C. Filtering on bounding boxes

Following the first results with train and finetuning of our dataset, we then decided to try an even greedier approach

TRAIN SETTINGS	SOLVER MAX_ITER	SOLVER BASE_LR	SOLVER STEPS	ROI_HEADS BATCH_SIZE_PER_IMAGE	SOLVER IMS_PER_BATCH
pretrained_untrained (default)	270000	0.02	[210000,250000]	512	16
pretrained_finetuning Synth	100000	0.00125	[45000]	128	1
pretrained_finetuning Coco	100000	0.00125	[90000]	128	1
pretrained_finetuning Mix	100000	0.00125	[90000]	128	1
scratch_trained Synth	100000	0.02	[45000]	128	1
scratch_trained COCO	100000	0.02	[90000]	128	1
scratch_trained Mix	100000	0.02	[90000]	128	1

Table I
TRAINING CONFIGURATIONS

at filtering the real dataset. This was decided because when looking at some reference images after testing, we noted that the model trained with our synthetic dataset was almost completely unable to recognize persons of small size often present in the background of the COCO images. We opted to use the data of the bounding boxes present in the annotation file of our real training dataset to filter out persons smaller than a given percentage on the whole image. A rough estimate of how generally big were the avatars in our synthetic dataset was computed, and with that we removed the annotations for which the percentage taken by their bbox over the image was lower than this threshold. The estimate was computed at one percent. As before, this process was handled using python and the code can be found in the *"datasets/coco/bbox.py"* file.

IV. MASK R-CNN

For this project, an object detection library called Detectron2 was used. Developed in Pytorch by Facebook as the successor of Detectron, Detectron2 supports a range of tasks such as object detection with boxes and instance segmentation masks, human pose prediction, semantic segmentation, and panoptic segmentation. Moreover, it comes with many models, like Faster R-CNN, Mask R-CNN[4], RetinaNet, DensePose, Cascade R-CNN, Panoptic FPN, and TensorMask.

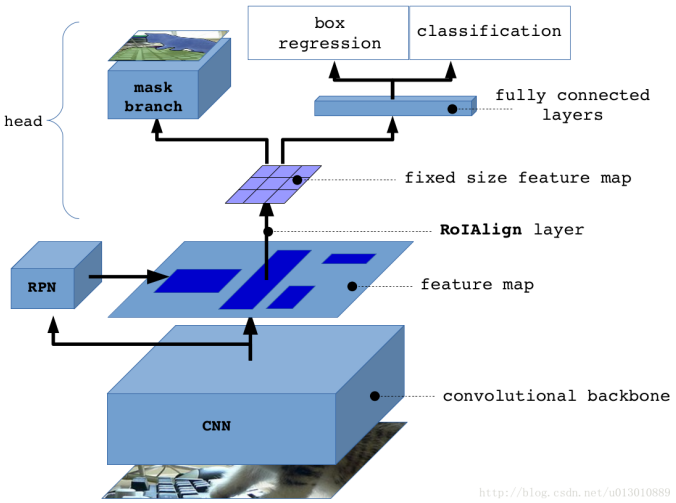
We relied on Detectron 2 for training, testing and visualizing the inferences made by our models. The COCO-pretrained R50-FPN 3x Mask R-CNN was selected as our baseline model (Fig.3). The convolutional backbone of the selected model is a ResNet+FPN backbone with standard conv and FC heads.

A. Training Phase

Being the synthetic annotated dataset we obtained so far in a COCO-like format, there is no need to do further processing before feeding it to Detectron2, as it has built-in functions to read it. In fact, to use such a dataset, it is enough to reference it by name after it has been registered with the function `register_coco_instances()`.

Then, it is required to use a model configuration that specifies the model and parameters to use. The configuration can be customized by changing its variables, and in our function `InitCfg()` it can be seen the parameters that we changed for our trainings, however the specific values are not necessarily those actually written in the script. Among the custom settings, the mask format must be set to "bitmask", as the masks in our dataset annotations are RLE encodings; and

Figure 3. Schematic showing the core elements of the used model, image from [5]



we had to limit the images per batch to one because of our hardware limitations.

Training, evaluation, and visualizing predictions are easily done with Detectron2, moreover, training data is logged and can be used in tensorboard to monitor the metrics at runtime, while the evaluation results are stored in a json and contain AP performances of both bbox and segmentation tasks. The full code of the described script is available in the *"detectron_model.py"* file.

In order to have a broader understanding of the quality of the synthetic dataset and the training itself, we prepared six distinct configurations, three that are different finetunings on the pretrained baseline model, and three based on the scratch untrained model. In addition, the untouched pretrained model served as our baseline. There is a difference between the the pretrained and from-scratch configurations that should be pointed out: they use different base configurations: the former ones are modifications of the *"COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml"* configuration file, while the others are derived from *"Misc/scratch_mask_rcnn_R_50_FPN_3x_gn.yaml"*. The two files, and the ones related to them, are included in the Detectron2 library, but we also made them available in the *run* folder of our repo for easy access. Table I contains the

TEST SYNTHETIC	BBOX						SEGMENTATION					
	AP	*AP50*	*AP75*	*APs*	*APm*	*APl*	*AP*	*AP50*	*AP75*	*APs*	*APm*	*APl*
pretrained_untrained (default)	64.394	89.338	75.378	45.380	69.067	78.852	58.808	88.316	72.592	37.761	62.104	73.981
pretrained_finetuning Synth	75.477	94.383	85.265	58.449	79.867	**86.733**	66.962	91.441	80.711	45.722	71.097	79.981
pretrained_finetuning Coco	65.289	90.921	76.851	47.384	69.696	78.205	60.372	90.113	74.571	39.735	63.698	74.952
pretrained_finetuning Mix	78.820	94.578	87.442	64.351	82.934	86.642	68.077	92.277	82.695	47.570	72.526	**80.177**
scratch_trained Synth	**79.891**	**95.401**	**88.549**	**65.556**	**83.980**	86.683	**72.162**	**92.287**	**84.025**	**53.463**	**76.561**	79.527
scratch_trained COCO	0.115	0.797	0.001	0.040	0.273	0.057	1.448	4.118	0.256	0.344	1.716	2.232
scratch_trained Mix	71.313	94.079	84.068	56.961	75.430	78.937	64.663	90.303	78.585	44.893	69.008	75.547
TEST COCO	**BBOX**						**SEGMENTATION**					
	AP	*AP50*	*AP75*	*APs*	*APm*	*APl*	*AP*	*AP50*	*AP75*	*APs*	*APm*	*APl*
pretrained_untrained (default)	54.449	85.037	**59.558**	41.294	**64.354**	**75.564**	45.675	81.374	47.210	31.296	**54.036**	68.103
pretrained_finetuning Synth	43.904	73.260	46.208	31.967	54.608	66.490	34.471	67.799	31.873	20.855	42.742	58.443
pretrained_finetuning Coco	**54.609**	**85.132**	59.513	**41.998**	63.311	75.004	**46.412**	**82.416**	**47.902**	**32.643**	53.889	**68.521**
pretrained_finetuning Mix	37.698	68.337	36.598	29.095	45.362	53.308	30.700	62.251	27.290	21.034	36.594	48.492
scratch_trained Synth	3.933	9.023	2.984	3.046	6.023	3.623	2.264	5.919	1.426	1.376	3.434	2.740
scratch_trained COCO	0.127	0.775	0.005	0.113	0.207	0.047	0.423	1.755	0.058	0.411	0.660	0.669
scratch_trained Mix	28.848	57.376	25.633	22.484	36.071	41.150	22.692	50.066	17.826	15.589	27.855	36.420
TEST COCO filtered	**BBOX**						**SEGMENTATION**					
	AP	*AP50*	*AP75*	*APs*	*APm*	*APl*	*AP*	*AP50*	*AP75*	*APs*	*APm*	*APl*
pretrained_untrained (default)	**67.826**	**94.620**	**77.490**	**41.217**	**65.173**	**73.938**	58.670	**92.663**	**67.765**	**23.011**	53.235	66.209
pretrained_finetuning Synth	57.106	87.518	62.628	24.005	53.271	64.174	46.489	82.720	48.208	8.992	40.956	55.326
pretrained_finetuning Coco	67.088	93.913	76.054	31.020	64.691	73.060	**58.712**	91.686	66.983	18.524	**53.329**	**66.283**
pretrained_finetuning Mix	43.883	77.505	43.251	13.192	42.328	49.503	36.226	70.319	33.751	4.745	31.533	43.821
scratch_trained Synth	3.455	8.545	2.605	1.453	4.007	3.113	1.945	5.043	1.452	0.159	2.140	1.977
scratch_trained COCO	0.061	0.386	0.003	0.002	0.132	0.090	0.201	0.912	0.016	0.003	0.483	0.862
scratch_trained Mix	32.356	65.403	28.227	9.478	32.815	36.381	25.841	56.220	21.747	3.087	22.969	31.340

Table II
TESTING RESULTS

training parameters used for each setup.

From the table it can immediately be seen that the difference in parameters between the pretrained baseline and our configurations is abysmal: comparatively, we did less iterations, used a smaller batch size per image, and had just an image per batch. This makes a head-to-head comparison unfair, but ultimately our goal is to understand the potential of synthetically generated data for developing models that behave well in real-world scenarios. As a side goal, it can also be shown the opposite, that is to verify that a model trained on real data such as COCO, can generalize well even against synthetic data.

Each train took about 7 hours on a single Nvidia Gtx 2060 super, with a batch size of 128 and 1 image for batch.

B. Testing Phase

In the testing phase, the evaluation was performed using Detectron2's functions, specifically, the COCO evaluator[6] was used, which evaluates AP (Average Precision) in the case of object detection and instance segmentation. For COCO, it is implied that AP is an average over all the categories, which is better referred as mean average precision (mAP), but in our scenario it they are really the same as there is just one category: person. Moreover, in its definition of AP for evaluation, COCO uses 101 recall thresholds, in the range [0:1] with steps of 0.1, and is averaged over multiple Intersection over Union (IoU) values, specifically 10 IoU thresholds ranging from 0.5 to 0.95, by steps of 0.05.

$$\mathrm{AP}_{\mathrm{COCO}} = m\mathrm{AP}_{\mathrm{COCO}} = \frac{\sum_{i=10}^{19} m\mathrm{AP}_{0.05i}}{10}$$

Averaging over IoUs averages rewards detectors with better localization, but for reference, more standard AP metrics with fixed thresholds are also computed: the AP at IoU=.50 metric used in PASCAL VOC, and a stricter version at IoU=.75. Finally, the AP score across different scales of segmentation masks areas are given; they are referred as AP^{small}, AP^{medium}, and AP^{large}.

Those metrics are computed for both the object detection (Bbox) and intance segmentation tasks, and the models performance are evaluated against three different datasets: a synthetic one ("Test Synthetic"), generated similarly to the training dataset using the Crowd Generator, and two real ones. Both of them are obtained from an official COCO testing dataset, but with some modifications. Our "Test COCO" is like the original COCO test, but it only has images containing at least 4 people, the idea being that this better aligns with our training dataset that is focused on portraying sparse crowds. Lastly, "Test COCO filtered" is an easier version of the former, where additional filtering removes people appearing too small in the scene. This last testing dataset is as close as we could get to the context of our training data, thus, it is expected that on this version our models trained on synthetic data perform reasonably well compared to the ones trained on COCO.

C. Results

Table II shows the performance of the trained models and the untouched pre-trained baseline across the aforementioned AP metrics, while a graphical representation is available in the appendix.

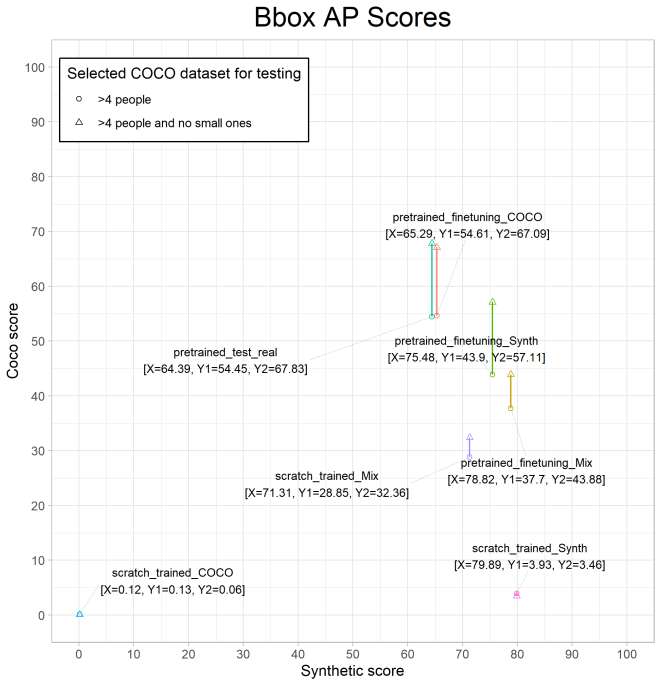


Figure 4. Object detection

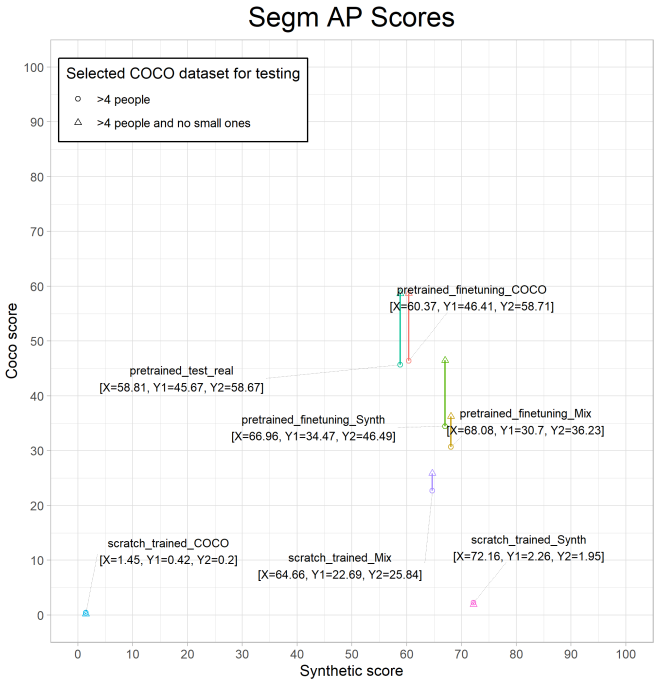


Figure 5. Instance segmentation

For a more effective understanding, we prepared two scatterplots, one for each task (Bbox and Segm), where each point is a training setup, and their position shows their AP performance across the performed tests. Specifically, the x axis gives the performance across the synthetic testing dataset, while the y axis represents the results on the COCO datasets. In order to distinguish between the two COCO datasets that were used for testing, we replaced each point of the training setups with two points, one for each dataset, and connected them to show the performance difference across the COCO variants.

In this kind of graph, a good model would be placed towards the upper-right corner because it should perform well on both synthetic and real scenarios. From this, it can be easily seen in Fig.4 and Fig.5, that our model trained from scratch purely on synthetic data (in purple), even if scored the highest in the synthetic test, did not achieve any good generalization for the real-world scenario of COCO, and more unexpected, it performed about the same on the simplified COCO dataset. On the contrary, it seems that the simplified one helped more the models pretrained on COCO.

A second surprise is given by the model *"scratch_trained COCO"*, that scored close to zero across all the three tests. The idea of this model was to see how close it would be to the pretrained COCO, given our training parameters. With similar enough parameters, the former would have performed close to the pretrained one. This effectively showed us how worse our chosen parameters were, and consequently how unfair the comparison is with the fully pretrained COCO.

Having seen that 100k iterations with batches of 128 images were not remotely enough for training COCO from scratch, unexpectedly, training a mixed version from scratch, that is using training images from both the COCO dataset and our synthetic one, has resulted to much better results (see *"scratch_trained Mix"*), despite the same number of iterations.

Moving on the three different COCO finetunings, *"pretrained_finetuning Coco"* is finetuning applied to our COCO dataset, where it should be recalled that it contains only images with at least 4 people. This version was expected to perform better than the original model on our COCO testing datasets, but it did not make a significant difference.

Finally, *"pretrained_finetuning Synth"* displayed the best trade-off between the real-world and synthetic scenarios if considering the performance on the simplified COCO dataset.

The considerations made so far are the same across the two tasks of object detection and instance segmentation, and the other AP metrics reflect the same situation. Still, the overview graph in the appendix better shows how much easier is the synthetic dataset with respect to the Coco ones, as all models (apart from the COCO trained from scratch) performed very well on the former. Among the models trained from scratch only the mixed one stands out for the real-world testing datasets.

Fig.6 shows some inference examples done by some of our trained models. We have chosen the most meaningful models to show the differences between a pure COCO, a pure Synthetic, and a mixed one.

V. Conclusion

If on one hand the parameters used in our training did not help to make a fair comparison with the pretrained COCO model, on the other hand this setting helped in emphasizing the strength of cross-dataset training. As the results have shown for

the case of *"scratch_trained_Mix"*, it performed unexpectedly well on both real and synthetic evaluations in comparison to *"scratch_trained_COCO"*, which it seems that did not have enough time to converge.

Secondly, *"scratch_trained_Synth"* also had time to converge with the same training parameters, showing that our synthetic dataset was maybe too simple and easy to train on. We integrated some functionalities to the simulator in order to increment the variance in the generated images, but we are conscious of the limitations of our setup. As a prove to this, it seems that this last model has overfitted and is not able to identify any objects in the real world images as we can see from Fig.6

Lastly, this experiment also confirmed us that the COCO model used as baseline performs well on synthetic data too; and with a bit of fine-tuning it can perform even better on it, but this is a trade-off with the performance on real world data.

For a more fair comparison, one could repeat the experiment with the same training parameters of the baseline model, but even without doing so it is clear from this that augmenting a real dataset with synthetic data can help with training and at the same time alleviates the burden of manually collecting and annotating large amounts of data. We can see the staggering difference in results between the *"scratch_trained_Synth"* and the *"scratch_trained_Mix"* by again looking at Fig.6, considering that the training was performed with the same settings. Ultimately, the deciding factor of the final result is the quality of the generator: while it is not easy to simulate realistic persons and environments, it is surely manageable to obtain an acceptable quality that would be beneficial in cross-dataset training. But if one wants to train purely from synthetic data, the "acceptable" threshold for good performing models is much more difficult to achieve.

A possible future work would be the retraining of our models using the same parameters of the baseline model, so that a fair comparison would finally prove or disprove our observations. While for what concerns our synthetic generator, there is room for improvement for sure: the UMA system has a lot of potential but the well crafted models and clothes available in the store are not for free, and needs a lot of time and effort otherwise.

REFERENCES

[1] umasteeringgroup, "UMA: Unity Multipurpose Avatar." [Online]. Available: https://github.com/umasteeringgroup/UMA

[2] Y. Wu, A. Kirillov, F. Massa, W.-Y. Lo, and R. Girshick, "Detectron2," https://github.com/facebookresearch/detectron2, 2019.

[3] "Crowd-Simulator." [Online]. Available: https://github.com/mmlab-cv/Crowd-Simulator

[4] K. He, G. Gkioxari, P. Dollar, and R. Girshick, "Mask R-CNN," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 2, pp. 386–397, 2020.

[5] L. Bienias, J. n, L. Nielsen, and T. Alstrøm, "Insights into the behaviour of multi-task deep neural networks for medical image segmentation," 10 2019, pp. 1–6.

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

Figure 6. Inference results

Models Performance Overview