User Manual for the DataSynth Pipeline

Hampus Arnestrand, Casper Mark

May 17, 2024

Introduction

This GitHub repository includes the unity project and Python code. The manual describes the functionalities and basics of using the synthetic data pipeline.

One of the most time-consuming parts of training an object detection model for a specific application is acquiring the training data. Thousands of varied images with bounding box annotations around the objects are necessary for a good model, traditionally obtained by manually taking pictures and then manually defining boxes for every single occurrence of the object in each image. This labeling process is additionally prone to errors when done by hand.

The first alternative to this is using an already available dataset of labeled images like Microsoft's COCO dataset of over 200 thousand labeled images. This won't work as soon as the model needs to be more specific. The next alternative that has seen an upswing in popularity in the last years is synthetic data. The images can be generated in a virtual environment built from 3D models. Through this approach total control of the dataset is gained depending on how we set up the environment and take virtual pictures. Another benefit is that every generated image can be automatically pixel-perfectly annotated from the 3D data.

The provided Python notebooks can then be used for converting the data to the right format, to be used for training a YOLOv8 object detection model.

1 Unity

The first thing to do after opening the Unity project is to check if it looks like in figure 1. If you can't see the GameObjects on the left (the lights, Main Camera, Volume, Scenario) that means that the DataSynth scene is not open. You can find it under the *Scenes* folder in the bottom left and then open it by double-clicking on it.

The unity project is called *DataSynth* and is built using the Unity Perception package. The process works by having a *Scenario* that calls on *Randomizers* which in turn randomizes most of the things in the scene. These can be found in the inspector after selecting the Scenario under the DataSynth scene on the left. Some of these Randomizers are built-in (documentation can be found under Unity Perception) and the rest are custom-made using scripting. *Randomizertags* can be added to different GameObjects and tells the Randomizers what objects to affect. Apart from the scenario, there is also a *Volume* (which controls for example the different camera effects, blur, etc.), *Main Camera* (which takes the images), and several *Directional Lights*. It is possible to change the number of lights by simply deleting and/or copying them.

1.1 Asset Tree

All the Project files are found in the Asset folder which is located under the Project tab in the bottom left corner of the screen. All the important folders will be described below.

1.1.1 Resources

The most important folder is the Resources folder. In this folder is a *ScriptableObject* file called *settings* which contains the parameters that can be controlled during the data creation. By clicking on it the settings should come up in the inspector tab on the right side of the screen. In this folder there is also the *3Dmodels* folder. All prefabs in this folder will be used in the creation of the synthetic data.

1.1.2 Distractor Objects

Here all the textures and models used for the distractor objects can be found, which includes background and occluder objects.

1.1.3 Scripts

All scripts can be found in this folder, for example, the custom Randomizers, Randomizertags, and other utilities.

1.2 How to create a prefab

After importing a 3D-model in FBX format you can drag it into the scene (left window) and then drag it back into the Asset tree. Unity will then give you a pop-up window where you will choose to create an **original** prefab. Make sure to delete the object from the scene! Now select the FBX model (not the prefab) and click the *Extract Materials* button under the *Materials* tab in the inspector (Tip: put the material in the **Materials** folder under **Resources**). Now you can assign whatever imported textures you want to the material. Lastly, you have to reassign the material to the prefab. Select the prefab and under $MeshRenderer \rightarrow Materials$ you can select the material. You can now see how the object will render at the bottom of the inspector while the prefab is selected. If you want the object to be included in the synthetic data creation, put the prefab somewhere in the **3Dmodels** folder. Otherwise, put it into the **Prefabs** folder. You can also put the textures in the **Textures** folder and the FBX file into the **Models** folder if you want to.

Note! If Unity gives you warnings about "animation clip" you can just ignore it.

1.3 Important Settings

To change the destination of where Unity places the created images and annotations go into $Edit \rightarrow ProjectSettings$. In the project settings you should find *Perception* on the left. In there you can change the **Base Path** to be your desired destination folder.

By clicking on the scenario, you can control the number of images created by changing the Iteration Count under *Fixed Length Scenario* in the inspector.

The resolution of the created images can be changed under the Game tab in the middle of the screen. This project can currently only support non-wide aspect ratios for the objects to fill the whole image, such as 4:3.

To start creating images, press the start symbol in the top middle of the screen. There you can also pause the process using the pause button. The third button can be used to create one image at a time after the process has been paused.

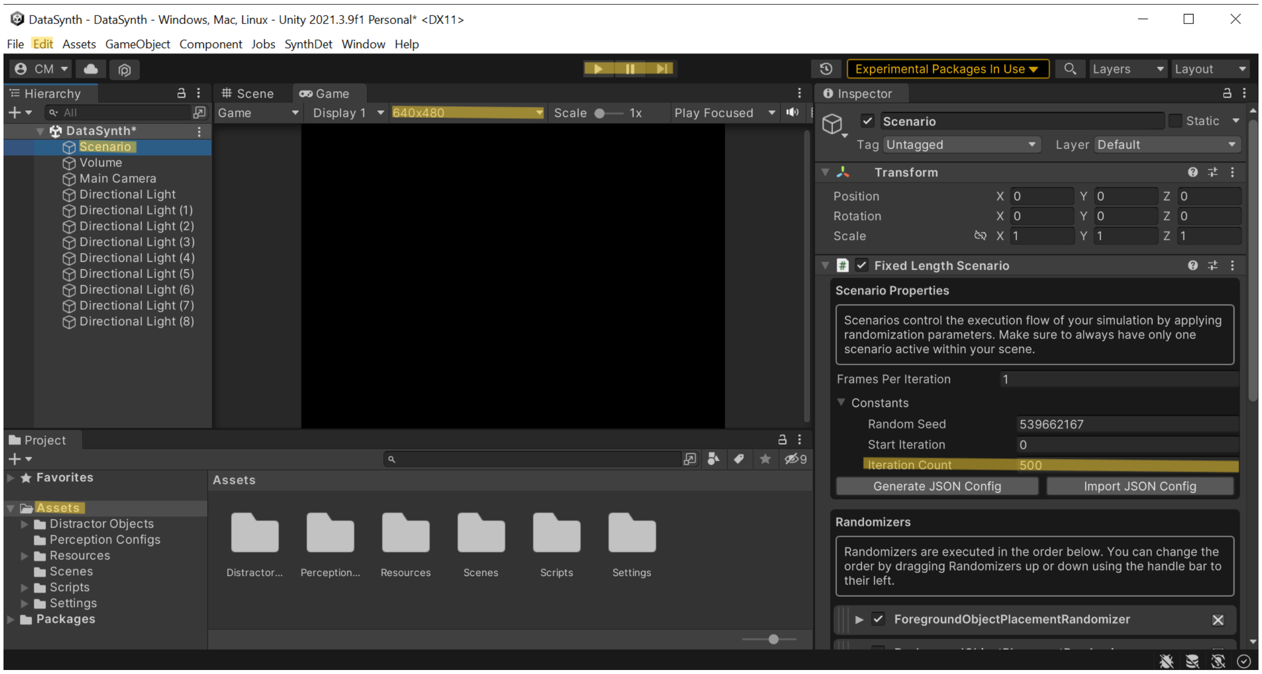

Figure 1: The unity layout where important tabs and fields have been highlighted in yellow.

2 Data Conversion and Training (YOLOv8.ipynb)

This Python notebook converts the annotated data from Unity to a format that can be used for the training of a YOLOv8 model. It prepares the .yaml file with instructions including the location of the converted data images and annotation .txt files, and the classes with respective IDs for labeling. Background photos of the real world are also included in the prepared dataset.

Before running the notebook you need to define the four following variables:

- **solo_folder** - String containing the path to the source folder where the data is located. It should end with 'solo' unless you have moved the data after its creation.
- **dst** - String containing the path to where the destination folder should be placed. This is where the converted data will be placed. The script will try to create the folder if it doesn't already exist.
- **background_photos** - String containing the path to the folder where the real background photos are located.
- **dataset_name** - The chosen name of the current dataset and model.

Several arguments can be changed, some of which are described below.

2.1 convert_format_perception_to_yolo

- **train_cutoff** - The number of images used for training. (Recommended to be a rounded fraction of *dataset_size*, a quantity that's detected automatically.)
- **val_cutoff** - The validation data cutoff. (With *dataset_size* as input, all images but the training images will be used for validation. A lower value will put aside the remaining images for testing.)
- **copy_images** - Boolean for choosing whether to copy or move images to the destination folder. (Moving the images results in a much faster runtime but removes them from the solo folder.)
- **minVisibility** - The minimum fraction of visible pixels for each object to be annotated.

2.2 include_bg_photos

Real background photos are important in the training for the model to be less prone to false positives. These photos should not include the objects trained on, therefore they don't need labels.

- **desired_fraction** - The desired fraction of real background images found in the final dataset. Around 5% background images seem to work well as a default. (If the folder with background images includes fewer images than needed, all available images are used.)

2.3 model.train

After the dataset preparation, the model training can be started, and the resulting weights can be used to export the model in another format (for example onnx) if needed.

The training can be thoroughly customized, though mostly keeping the default arguments results in a good model. The following arguments could need to be changed depending on the use case. To find all the arguments that can be used, see the documentation on Ultralytics website.

- **epochs** - Depends on the number of classes, the dataset size, and more. (As a reference, with just a few thousand images and one class, it can be enough with 30 to 50 epochs.)
- **imgsz** - Defines the image resolution by the largest side pixel count.

3 Inference (YOLO_inference.ipynb)

The separate notebook for inference includes code cells that can run inference on given images, videos, or video streams.

Validation of the model can be done if an annotated test dataset is provided. The code then prints out statistics such as recall, precision, mAP50, and mAP50-95. Ultralytics also automatically saves images of the predictions, as well as graphs of the results where the Python script is located under a folder called *detect*.

4 Hailo

For the model to be used with the Hailo accelerator chip a model conversion and optimization is needed. Using Hailo model zoo, the following command can be run for a basic conversion from the onnx format:

```
hailomz compile --ckpt yolos_model.onnx --calib-path calibration_images --
    yaml hailo_model_zoo/cfg/networks/yolov8s.yaml --classes 1
```

The calibration images can for example be the training images.