RBE 474X
Project 1

Colin Balfour
Worcester Polytechnic Institute
Worcester, Massachussetts 01609

Khang Luu
Worcester Polytechnic Institute
Worcester, Massachussetts 01609

Thinh Nguyen
Worcester Polytechnic Institute
Worcester, Massachussetts 01609

I. Part 1

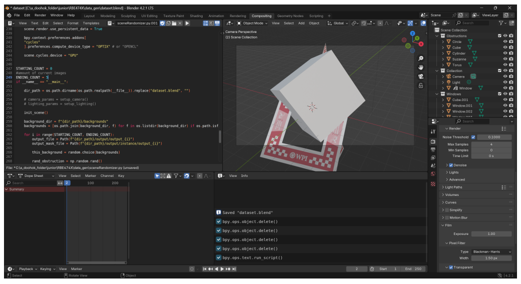
A. Dataset generation

We used Blender's Python API to generate a dataset of 5000 RGB images with 640x360 pixels. Each RGB images also has a corresponding instance segmentation mask, which is a grayscale image with the same dimensions, and different grayscale value for different instances.

Windows are generated randomly, with a random number of windows placed in the image, and a random scale for each window. The windows are placed randomly in the image. The background is picked randomly from a dataset of 10000 background images, in different lighting, indoor or outdoor, etc. The instance segmentation mask is generated by assigning a unique grayscale value to each window. Obstruction is picked from 8 preset objects in blender, and placed randomly to obstruct the view of the windows.

Hyperparameters for the dataset generation: - Min and Max number of windows: 1 to 3 - Min and Max object scale: 0.25 to 3 - Min and Max object rotation: 0 to 3/2pi - Min and Max object translation: -3 to 3 meters - % of images with an object obstructing a window: 50% - Obstruction: 1 object, randomly picked from 8 preset objects in blender - rotation: 0 to 3/2pi - translation: -1.5 to 1.5 meters - scale: 1.5 to 2

Obstruction objects are placed in an "Obstruction" collection, out of the camera view, and then copied into the view. The original window is also done the same. Objects copied in (the obstruction and windows) are in a "window" collection, and are deleted and re-copied every image iteration.



```
* dataset [C:\a_doohok_folder\junior\RBE474X\data_gen\dataset.blend] - Blender 4.2.1 LTS
```

```
    scene.render.use_persistent_data = True

    bpy.context.preferences.addons[
    "cycles"
    ].preferences.compute_device_type = "OPTIX" # or "OPENCL"

    scene.cycles.device = "GPU"


STARTING_COUNT = 0
#amount of current images
ENDING_COUNT = 5
if __name__ == "__main__":

    dir_path = os.path.dirname(os.path.realpath(__file__)).replace("dataset.blend", "")

    # camera_params = setup_camera()
    # lighting_params = setup_lighting()

    init_scene()

    background_dir = f"{dir_path}/backgrounds"
    backgrounds = [os.path.join(background_dir, f) for f in os.listdir(background_dir) if os.path.isf

    for i in range(STARTING_COUNT, ENDING_COUNT):
        output_file = Path(f"{dir_path}/output/output_{i}")
        output_mask_file = Path(f"{dir_path}/output/instance/output_{i}")

        this_background = random.choice(backgrounds)

        rand_obstruction = np.random.rand()
```

```
File: *C:\a_doohok_folder\junior\RBE474X\data_gen\sceneRandomizer.py (unsaved)
```

```
Saved "dataset.blend"
bpy.ops.object.delete()
bpy.ops.object.delete()
bpy.ops.object.delete()
bpy.ops.object.delete()
bpy.ops.text.run_script()
```

Fig. 1. Blender GUI for dataset generation

While the dataset generation is slow, and obstruction is not always placed in a realistic way and not textured, the dataset is diverse enough to train a model to generalize to detect windows in images.

Please note that the dataset is not included in the submission. To run the dataset generation script, please make sure output directories in the python file are set correctly.

B. Results

II. Part 2

A. Implementation

B. Results

III. Part 3

A. Implementation

In Part 3, we compared the performance of the MLP implemented in Part 1 with a more modern CNN architechture, using PyTorch's inbuilt layers for simplicity. The model was trained on the same CIFAR-10 dataset with 40 epochs, also using Adam Optimizer with a learning rate of 1e-4 and a batch size of 32. The training loss curve and confusion matrices for the validation set at the first and last epochs are shown in figures 4 - 6.

B. Results

The CNN model performed significantly better than the MLP, achieving an accuracy of 56.79% compared to the MLP's 43.68%. This is due to the CNN's ability to learn spatial features in the images, due to the convolutional layers, where the network learns a kernel filter to apply to the image to extract such features useful in classifying images. The MLP, on the other hand, does not have such structure, and must learn the features from the raw pixel values, which is much more difficult. It also was faster to train, with the CNN taking about 7m 28s to train, compared to the MLP's 8m 36s. This is likely due to efficiencies in PyTorch's optimization compared to our custom implementation. CNNs can be more efficient than MLPs, due to the convolutional layers reducing dimentionality (and therefore the number of parameters), but in this case the CNN also had more layers/parameters, so it's unlikely the network architecture contributed to its efficiency

IV. Part 4

A. Implementation

Part 4 implemented custom layers for a Convolutional Nerual Network (CNN), again cuda-optimized with PyTorch. Like in Part 1, forward and backward passes were implemented for a Convolutional (Conv2d) layer. This included an implementation of a 2d cross-correlation method, which is the main operation performed by the Conv2d layer. The forward and backward propogations were implemented for any number of input channels, output channels, kernel size, stride, and batch size. The custom implementation was then validated against PyTorch's built-in layers.

B. Results

All implementation yeilded the same results as with PyTorch's built-in layers. The output is in the jupyter notebook.

V. Part 5

Part 5 was not attempted, despite the custom CNN implementation, due to time constraints, both in the code and in the humans writing it.