RBE 474X: Dramatic Data

Jakub Jandus, Hudson Kortus, Dexter Stark
Worcester Polytechnic Institute
Worcester, Massachusetts

***Abstract*—In this project, we learned how to use a 3D modeling software called Blender in order to simulate training data for a neural network. By using simulated data as opposed to real data or pictures, we are able to train the network with a much larger amount of data as well as using data that may would be either impossible or dangerous to recreate in the real world.**

I. Introduction

For project P2: Dramatic Data, the goal was to simulate a large amount of training data for a network by using the 3D modeling software called Blender. By using simulated data, the hope is that we would be able to train the network to recognize drone racing windows at various angles and distances, regardless of how many were present. To meet the criteria of the project, we needed to generate approximately 50,000 images in order to train the network to be as accurate as possible.

II. Blender

Collecting and segmenting 50,000 images is not feasible by humans, so simulated data is necessary. Blender offers a few different options that are capable of generating the images we needed, after exploring options like BlenderProc we settled on using the default Blender scripting. After learning the basics and how to apply textures to objects, we focused on using Blender Script to create random scenes that would include up to four windows and the occasional obstacle. The number of windows, chance of obstacle occlusion, and distance were controlled probabilistic variables to allow us to effectual generate a diverse dataset. The windows were rendered with no background to save on rendering computation and image size. Blender was tasked with rendering 5,000 images and then a separate program added randomly rotated background images.

Image blurring, Gaussian-noise, and color shifts are randomly applied by the network data loader to further save on compute and image storage. When the trainer requests an image from a set of 50,000, the first 4 digit places (ten-thousands, thousands, hundreds and tens) were used to specify which image should be used from the set of 5,000. The ones place was used to map to a random set of the image augmentations. While this minimally increased computation it allowed the group to only generate and store 5,000 images. The rendered output with a rotated background and random augmentation are shown alongside the segmented "ground truth" generated by Blender in figures 2, 3, 4.

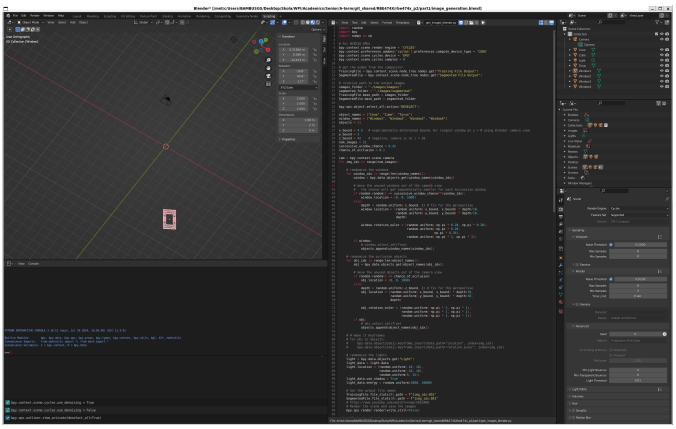

Fig. 1. Our Blender GUI

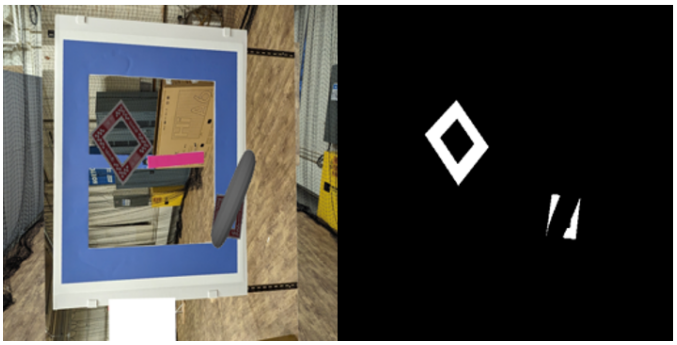

Fig. 2. Generated Data and Label #1

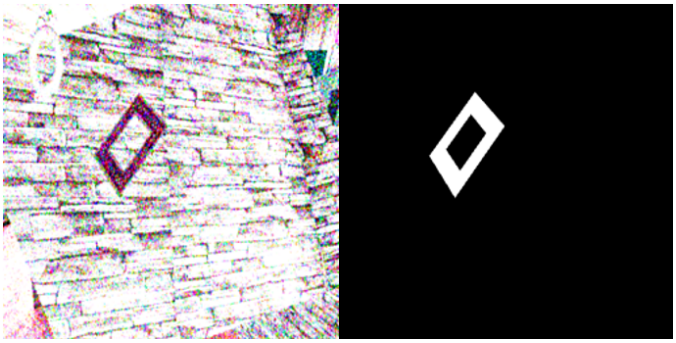

Fig. 3. Generated Data and Label #2

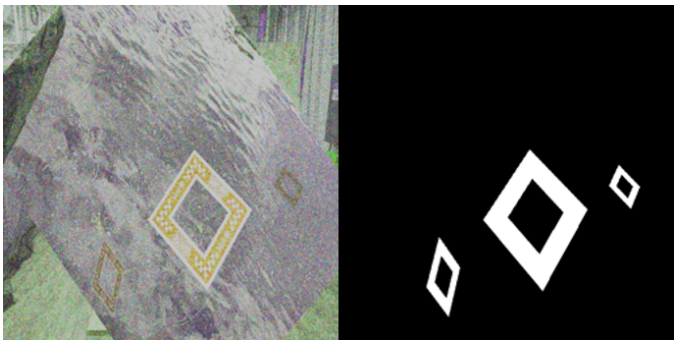

Fig. 4. Generated Data and Label #3

III. U-Net inspired CNN

The group wanted an to combine the image segmentation capabilities of a simple encoder decoder neural network with the flexibility and training speed of a ResNet. After some research the group came upon a simple U-Net inspired architecture [1], shown in figure 5. A U-Net features a downward section contracting where images are shrunk using max pooling and channels doubled during convulution, a bottleneck section with a small image size and numerous channels, and an upward section where images are put through de-convolution (transposed convolution). Layers on the upward section of the network are concatenated to the downward equivalent in order to restore some context and speed training.

The model was implemented using python in a format that is easy to use by a learning programmer. Training was performed overnight using the Turing cluster on a A30 GPU. Training progress was tracked using Wandb and resulting graphs of the training and validation loss over epoc are shown in figures 6,7,8. As shown accuracy rapidly improves and then flattens out. We found there were minimal differences between the model trained to 40 epocs and 100 epocs. Nevertheless, the U-Net architecture prevented us form over fitting the model. The hyper parameters were tuned over multiple trainings to get the best results are shown in table I.

Hyper-Parameters	Value
Learning Rate	5e-6
Batch Size	8
Optimizer	AdamW w/ Weight decay of .001

TABLE I
TABLE OF TUNED HYPER PARAMETERS USED TO TRAIN NETWORK

IV. Loss Function Selection

The loss function of a neural network is a means of determining how well it is able to model a dataset. A popular choice is the CrossEntropy, as it is capable of categorizing multiple classes within the same image based on probabilities of each pixel belonging in a classifier class. As useful as this is, it exceeds the needs of this project and could decrease the convergence rate. A suitable alternative is to use a Binary Cross Entropy (BCE) loss function, which is meant to focus on identifying only one class within a given image. While this is

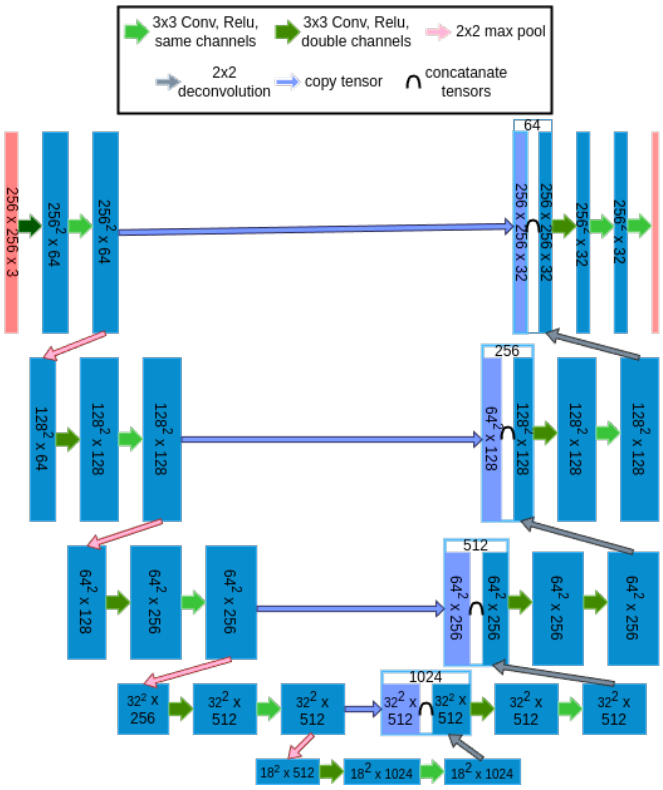

Fig. 5. Network Architecture Diagram

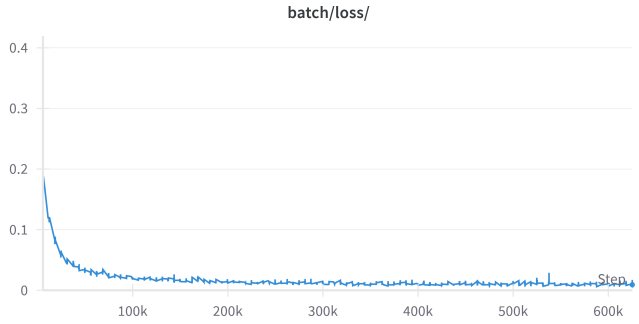

Fig. 6. Loss Accuracy

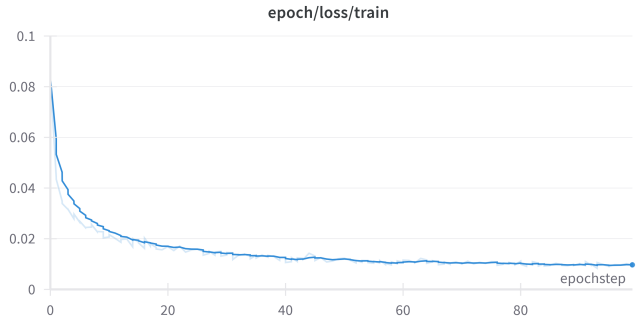

Fig. 7. Training Loss Accuracy

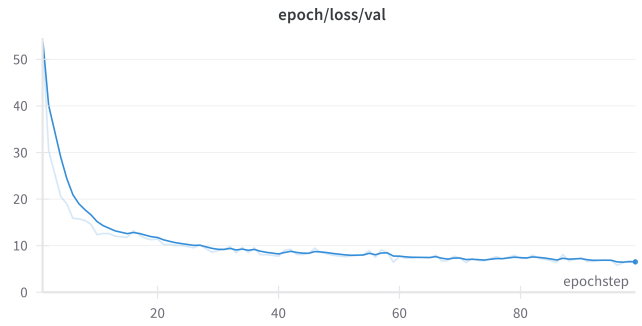

Fig. 8. Validation Loss Accuracy

better for what we are trying to accomplish, this loss function in Pytorch can only take probability numbers – numbers between 0 and 1 – as an input. Since we are passing in the raw network output (logits), our final choice ended up being a variation of the BCE loss function with logits that has an integrated SoftMax pass before calculating the probabilities.

V. Failure Cases

When doing our initial tests on the final video, we found a few faults in our network that we decided could be fixed for future versions. The first issue we noticed was that when window takes up a large majority of the screen, it would have a difficult time keeping its image clear and consistent while in motion. The other issue being that when the fire extinguisher becomes into frame, it is quickly labeled as one of the windows.

Fig. 9. Failure Case: Fire Extinguisher

Fig. 10. Results From Final Video

After discussing possible reasons for why these mistakes were taking place, we concluded that it must be because the network had been much more focused on identifying the shape and color of the window, while placing less emphasis on the pattern that the window has. The extinguisher is potentially an exception to this, as it may be recognizing the alternating red and white on it as being the window's pattern. A few potential fixes we could use to prevent this in the future would be to increase the amount of data where the windows are rendered closer to the camera, using a no padding method to give the kernels more impact, and to use dilated convolution.

Fig. 11. Sample Training Image

Fig. 12. Results of Training (Clean). From left to right: Ground truth, Source Image, AI Output

Fig. 13. Results of Training (W/ Filters and Noise). From left to right: Ground truth, Source Image, AI Output

Fig. 14. Results W/ Color Filters. From left to right: Ground truth, Source Image, AI Output

Fig. 15. Results W/ Overlap, Noise, and Color Filters. From left to right: Ground truth, Source Image, AI Output

VI. Conclusion

The use of simulated data to train a neural network is incredibly helpful and has various benefits. Through this project we learned that we are able to produce an incredibly large amount of data that can perfectly suit the needs of the user. By augmenting the placement of the desired object, backgrounds, filters, and lighting, we are able to customize the network to focus on specific points of interest and fine tune it to give the desired results. In addition to this, it has taught us that we are capable of creating data that would otherwise be too difficult to reproduce on a larger scale. The trained model can be found here.

Acknowledgment

References

[1] Ronneberger, O., Fischer, P., & Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation (arXiv:1505.04597). arXiv. https://doi.org/10.48550/arXiv.1505.04597