Dramatic Data!

Caleb French
School of Robotics Engineering
Worcester Polytechnic Institute
Worcester, Massachusetts 01609
Email: cmfrench@wpi.edu

Sukriti Kushwaha
School of Robotics Engineering
Worcester Polytechnic Institute
Worcester, Massachusetts 01609
Email: skushwaha@wpi.edu

Chad Nguyen
School of Robotics Engineering
Worcester Polytechnic Institute
Worcester, Massachusetts 01609
Email: ctnguyen@wpi.edu

***Abstract*—This report contains the results of creating a convolutional neural network to semantically segment frames trained from simulated data. Data was generated from Blender and augmented to create a large enough dataset to train a U-Net network for segmenting frames from the background.**

I. Introduction

The purpose of this lab was to segment PeAR racing windows from the backgrounds of a video with either one or multiple windows in frame at a time. To accomplish this goal, the following tasks were completed:

1) Generate renders of PeAR racing windows in Blender with random angles, amounts of windows, and lighting
2) Use homography transformations on the Blender renders to generate unique angles of the windows
3) Use image augmentation to generate unique filters over the windows to create larger dataset
4) Create U-Net to train for semantic segmentation task
5) Train U-Net to segment windows from a video

By following the above tasks, our neural network was effectively trained using the sim2real process of generating training data using blender and applying augmentations to create over 50,000 unique images. U-Net was then chosen as the convolutional neural network for semantic segmentation.

A. Generating Renders from Blender

Blender is a free, open source 3d Rendering platform that allows users to create and process virtual scenes. While Blender has a variety of different uses, from video production to game design, it also has a very handy application in data generation. Since neural networks need an incredibly large amount of data, and the scope of our project provides us neither the time nor funds to get real data to train our network, we can use blenders to generate scenes very close to real life.

1) Script for Randomization: First, to set up our scene we obtained a high-quality photo of the window and put it into our scene. Next, we varied our background by creating a cube that the camera and window were inside with the texture of some background images. To vary our lighting and camera angle, we then used a script that would use spherical coordinates to keep the camera and lighting above the scene, and randomly vary the distance and angle for both within a certain domain. We wanted to add obfuscation, so that our neural network would be able to identify the goal even if there were obstacles in front of them. So, we created some objects and would place them randomly in the scene so they would sometimes block the view of our window. Finally, we wanted our network to be able to identify all windows on screen, so we made two more windows that would sometimes appear in the scene, such that for any given image there could be between one and three windows. Along with the image itself, because we were generating all these images virtually, we were able to easily generate the respective ground truth by simply generating the alpha of the render. The only downside to blender is how much time each render takes. While it is certainly much faster than taking photos and marking them, the process still takes around 30 seconds per image, and we need tens of thousands of images for our Neural Network. We ended up generating around 460

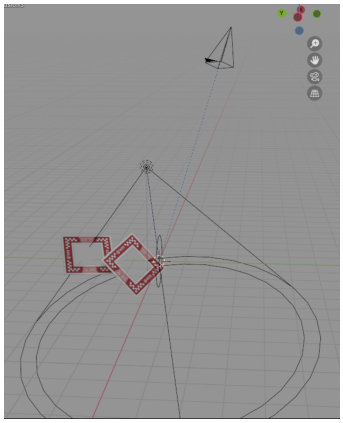


Fig. 1: Racing Windows In Blender

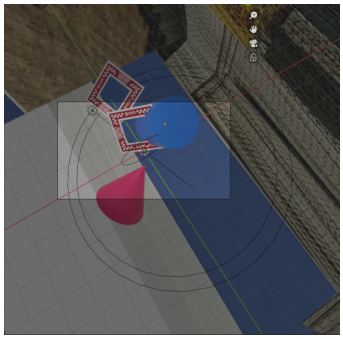


Fig. 2: Windows With Background and Occlusions in Blender

renders, and used Image Augmentation to very quickly create "new" renders from our existing ones.

Fig. 3: Alpha Generated from Blender

B. Image Augmentation

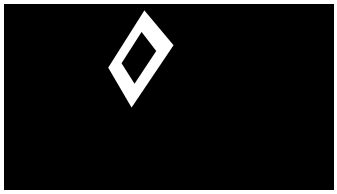
Image augmentation is integral for the sim2real process since generating datasets are extremely time consuming, image augmentation can be utilized. A single blender render can take up to 30 seconds per image when with image augmentation, unique images can be generated in up to 0.25 seconds. Due to the magnitudes of speed, image augmentation is integral for creating a large dataset.

1) Homography Transformations: Homography transformations were generated using the torchvision function *perspective(img, startpoints, endpoints)*, where img is the image to augment, startpoints are the four corner coordinates of the image to augment, and endpoints are the four corner coordinates of the output image. The four point pairs are then used to solve the homography transformation frame which is applied to the entire image cuasing the window to appear in different perspectives. For each render generated from blender, 7 unique homography frames are applied. As seen in Figure 4 and Figure 5, both the original render and label are transformed with the same homography matrix to create a unique frame.

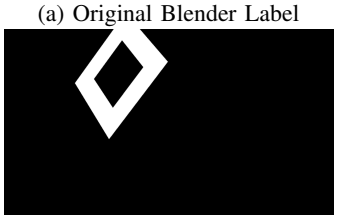
(a) Original Blender Render

(b) Homography Transformation Render

Fig. 4: Renders Of Frame 70



(a) Original Blender Label



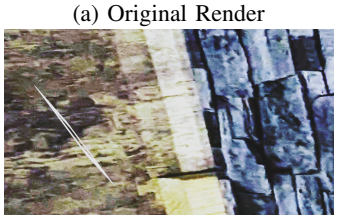
(b) Homography Transformation Label

Fig. 5: Labels of Frame 70

2) Torchvision Augmentation: With the goal of generating at least 50,000 images for training, generating from blender and homography transformations are not enough unique images. Also, to create a diverse enough dataset for training, images cannot be perfectly generated since it is unlikely for real collected data to be pristine. Therefore, data augmentation is used to generate more unique and diverse images. For each render, including the original and the transformed ones, 15 augment frames were created.

To diversify the types of augments that are being applied to each image, each augmented image had a random chance for either 1 filter or 3 filters to be applied. Once the amount of filters are chosen, there is a random choice between 6 filters to be applied. The 6 filters are: Gaussian Blur, Augmenter with Gaussian Blur, Color Shifter, Posterizer, Solarizer, and Equalizer. The augments create blur, color changes, and artifacts to mimic realistic noise from real data collection. With the dataset now established, it was time to create the neural network for semantic segmentation.

(a) Original Render



(b) Augmented Render

Fig. 6: Before and After Augmentation

C. U-Net Creation

For semantic segmentation, the U-Net structure was used to create our deep neural network. U-Net is a Convolutional Neural Network that mainly consists of a Encoder-Decoder structure [1]. The Encoder and Decoder mirror each other in which the Encoder uses convolutions making the outputs smaller and the Decoder uses deconvolutions to make the outputs larger. By the end, the output image is similar sized to the initial input. Each block in the Encoders and Decoders are made up of 6 layers: convolution, batch normalization, ReLU, convolution, batch normalization, and ReLU. Then the outputs are max pooled and then ran through the same block structure. This was done 4 amount of times. The number of features corresponds to the number of output channels, which was doubled during each encoder block. At the same time, there is a skip connection that connects from the Encoder to the Decoder. The entire architecture can be visualized in Figure 7.

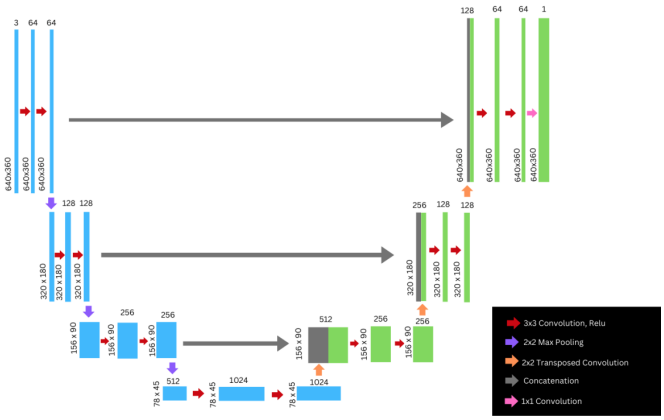


Fig. 7: Architecture of UNet

II. Conclusion

The loss function used for training was Binary Cross Entropy with Logits Loss (BCEWithLogitsLoss). BCEWithLogitsLoss was used since it combines a sigmoid layer with BCELoss to create a more stable loss function. BCEWithLogitsLoss is also more effective with segmentation tasks.

The failure cases that we ran into were segmenting when the camera moved up close to the racing window, and often having large holes in our segmentations as shown in Figure 12. The issues when the camera was close to the windows is likely because the rendered frames generated in blender did not have enough close up views of a single frame. Hence, the model was able to perform well at a distance from the racing window but missed some sections of the window up close. As for the large holes, our occlusions were very often included in the renders (70% of the time), and would often cover large portions of the windows. This is not what happens in the video, as most occlusions were either the windows themselves or very small objects (such as the post-it notes). It is likely due to this that the windows would be biased to generate with large holes in them.

The optimizer used for training was the ADAM model. We re-used ADAM due to its effectiveness in the previous lab. ADAM is an effective optimizer because it finds the local minima but also has variance in each step allowing it to not optimize around a small local minima. The learning rate used was 10^{-4}. The number of epochs ran was 20, with epoch 14 having the highest validation. The batch count was around 3,900, with a total of 59,000 images.

These are some of the segmented images generated from the validation set:

Fig. 8: Segmented window in presence of occlusion

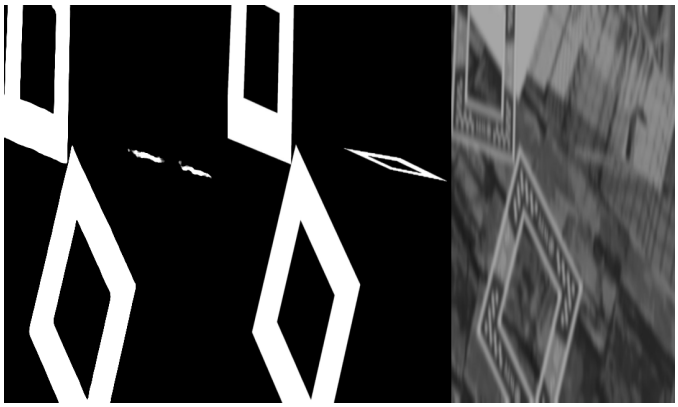


Fig. 9: Multiple Window Segmentation

Additionally, these are some of the segmented images generated from the test video:

As seen from both the validation and test video, the validation tests were more accurate than the segmentation on the video. This is expected since the validation data was also generated in the same way that the training data was formulated but the test video data was completely different.

As seen in Figure 14, the training loss curve per batch shows that the model continued to perform better and lower loss per batch. Additionally, in Figure 14 the validation loss curve per epoch shows that generally performed better with more epochs. When comparing the two curves to each other, it is clear that the training set continued to get better when the validation set did start to plateau. This would be due to over

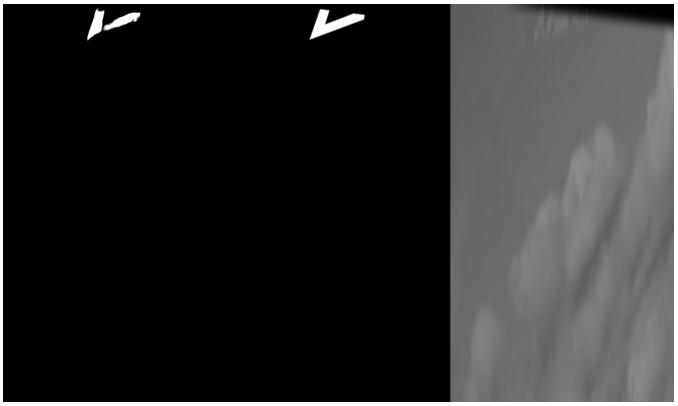


Fig. 10: Blurred Window Segmentation

Fig. 11: Single Window Segmentation

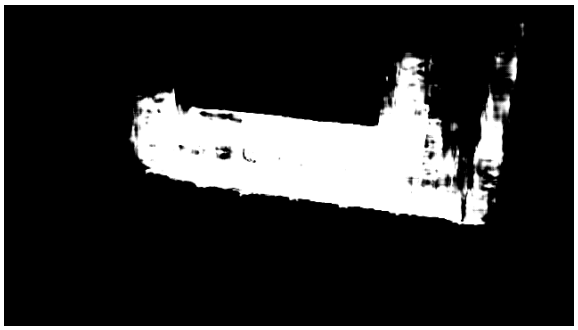
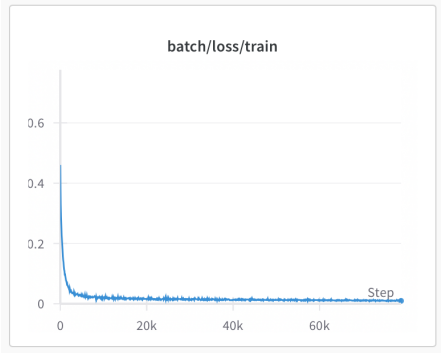


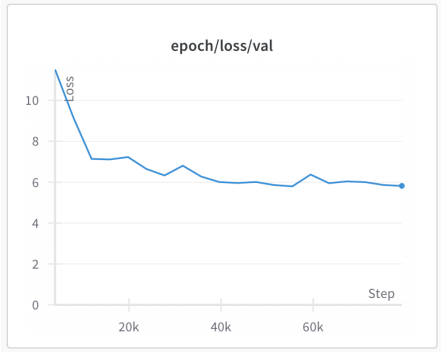
Fig. 12: Close Up Single Window Segmentation

Fig. 13: Multiple Window Segmentation

fitting to the training data making the model perform worse on new images.



(a) Training Loss Curve per Batch



(b) Validation Loss Curve per Batch

Fig. 14: Training Loss Curves

The best checkpoint found for the deep neural network was Epoch 14, with a link to the .pth here: Epoch 14 .pth

ACKNOWLEDGMENT

The authors would like to thank Professor Nitin Sanket for teaching us all of the necessary material to complete such a project. The authors would also like to thank Manoj Velmurugan and Phillip Brush for grading our project and providing assistance in the use of the Turing Cluster and understanding the components of an Encoder-Decoder architecture.

REFERENCES

[1] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation." Available: https://arxiv.org/pdf/1505.04597