

VIP Lunabotics Senior Design Report

Nikita Ravi

Abstract—The goal of this senior design project is to support the Lunabotics club in their effort to develop an autonomous mining robot to compete in a NASA lunar-style mining competition. My contribution to making this goal one step closer to reality was to develop a pipeline that would transform a front-facing photo captured by the rover to a top-down view. This continuous transformation of photos would enable the development of a GPS mapping system. In addition to the pipeline, I implemented RGB-Depth reconstruction to provide the option of creating a custom dataset to train the pipeline on. Source code for this senior design is available at: <https://github.com/nikkiravi/VIP-Lunabotics-Software-Senior-Design>

I. INTRODUCTION

This design project initially began with conducting extensive background research on the subject matter such as understanding what deep learning in computer vision entails, the different focus areas of computer vision: Object Detection, Object Tracking, Masking, Stereo Vision, and Noise Removal. In addition to the subject matter, I also studied different Graphing Algorithms like Dijkstras, A*, D* algorithms and as well as Sampling Based Algorithms like PRM, PRM*, RRT, RRT* algorithms to prepare for the possibility of switching from a computer vision focused project to a motion-planning project.

Once I completed the research phase, I found myself finding the computer vision (CV) aspect of robotics more interesting and therefore, decided to proceed with this focus area in my second semester of senior design with the objective of developing a pipeline that would transform a front-facing view to a top-down view of the lunar environment.

The implementation phase of the CV pipeline first began with finding two research papers on deep learning techniques for deblurring an image and its transformation from a front-facing view to a birds eye view (BeV) perspective. The deblurring research paper I decided to re-implement is called *DeblurGAN-v2: Deblurring (Orders-of-Magnitude) Faster and Better* [1]. The BeV transformation research paper I decided to re-implement is called *A Sim2Real Deep Learning Approach for the Transformation of Images from Multiple Vehicle-Mounted Cameras to a Semantically Segmented Image in Bird's Eye View* [2]. In order for the deblurred image to be passed as an input to the BeV deep learning network, the image first needs to be semantically segmented which is where training the detectron2 model comes in. Finally, I implemented RGB-Depth reconstruction to provide the option of creating a custom dataset to train the pipeline on.

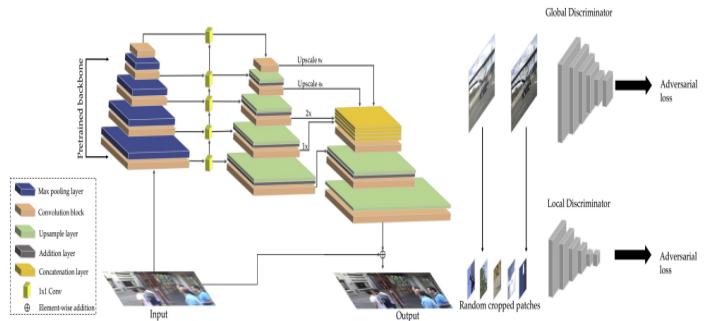


Figure 2: DeblurGAN-v2 pipeline architecture.

Fig. 1. Architecture of DeblurGAN-v2 [1]

II. IMPLEMENTATION OF DEBLURRING

A. Architecture of DeblurGAN-v2

1) *Feature Pyramid Deblurring*: Processing multiple scale images is time-consuming and memory demanding which is why the paper introduces Feature Pyramid Networks (FPN). The FPN module generates multiple feature map layers which encode different semantics and contain better quality of information [1]. FPN comprises a bottom-up and a top-down pathway where the bottom-up pathway is a CNN for feature extraction, with the spatial resolution being down-sampled which enables more semantic context information to be extracted and compressed [1]. Through the top-down pathway, the FPN reconstructs higher spatial resolution from the semantically rich layers [1]. The connections between the bottom-up and top-down pathways adds high-resolution details and help localize objects [1].

2) *The Backbone*: The FPN module is agnostic to the choice of feature extractor backbones. **Inception-ResNet-v2** was used to pursue a strong deblurring performance over **MobileNet V2** and **MobileNet-DSC** since we are not working with mobile-on-device image enhancement. The Inception-Resnet-v2 network has made great strides in image recognition performance due to it being a combination of the *inception* network with residual networks [3].

3) *Double-Scale RaGAN-LS Discriminator*: GAN stands for Generative Adversarial Networks and it consists of two models: a discriminator D and a generator G , that form a two-player minimax game [1]. The generator learns to produce artificial samples and is trained to fool the discriminator, with the goal of capturing the real data distribution [1]. Since an ordinary GAN is difficult to optimize, the Least Square

GANs (LSGAN) discriminator introduced a loss function that provides a smoother and non-saturating gradient [1]. This log-type loss saturates quickly as it ignores the distance between x to the decision boundary [1]. On top of this, the paper adopts relativistic wrapping on the LSGAN cost function, thus creating **RaGAN-LS** loss. This allows the training to go much faster and the authors have empirically concluded that the generated results possess higher perceptual quality and overall sharper outputs [1].

4) The Training Algorithm: While training the networks on the blurred images, we need a metric that would assess how close the outputs of the network is to the original image. This metric or cost function is a combination of *pixel-space loss* L_p , *perceptual distance* L_x , and L_{Adv} containing both global and local discriminator losses, as shown below.

$$L_G = 0.5L_p + 0.006L_x + 0.01L_{Adv} \quad (1)$$

B. Training Datasets

For the purpose of training the DeblurGAN-v2 model, I used the **GoPro** dataset which uses the GoPro Hero 4 camera to capture 240 frames per second (fps) video sequences and generate blurred images through averaging consecutive short-exposure frames [1].

C. Implementation

The following table shows the parameters that were used to train the DeblurGAN-v2 network. The parameters are identical to the author's implementation to make an attempt at replicating their results.

Parameters	Value
Epochs	100
Learning Rate	0.01
Optimizer	Adam
Train Batch Size	1
Train Batch per Epoch	1000
Validation Batch per Epoch	100
Generator	FPN Inception
Discriminator	Double GAN

TABLE I
THE PARAMETERS USED TO TRAIN DEBLURGAN-V2

D. Results

Figure 2 shows the results of my training for the last few epochs out of which these are the best results:

- Best Validation G_{loss} : -1.3448
- Best Validation PSNR ¹: 24.6236
- Best Validation SSIM ²: 0.8173

¹Peak Signal to Noise Ratio is used for quality estimation for the loss of quality [4]

²Structural Similarity Index measures the similarity between two images [5]

```

Epoch 149, 38 0.06551556375; 1000 [00071000 1110340000, 1.341575, loss=0.1427] PSNR=23.765; SSIM=0.7724
Epoch 150, 38 0.06551556375; 1000 [00071000 1110340000, 1.351575, loss=0.1327] PSNR=23.0775; SSIM=0.8148
Validation: 1000 [00071000 1110340000, 1.361575, loss=0.1227] PSNR=23.2199; SSIM=0.8352
Epoch 151, 38 0.06551556375; 1000 [00071000 1110340000, 1.371575, loss=0.1127] PSNR=23.3305; SSIM=0.7743
Validation: 1000 [00071000 1110340000, 1.381575, loss=0.1027] PSNR=23.4411; SSIM=0.8581
Epoch 152, 38 0.06551556375; 1000 [00071000 1110340000, 1.391575, loss=0.0927] PSNR=23.5517; SSIM=0.7838
Validation: 1000 [00071000 1110340000, 1.401575, loss=0.0827] PSNR=23.6623; SSIM=0.8892
Epoch 153, 38 0.06551556375; 1000 [00071000 1110340000, 1.411575, loss=0.0727] PSNR=23.7729; SSIM=0.8992
Validation: 1000 [00071000 1110340000, 1.421575, loss=0.0627] PSNR=23.8835; SSIM=0.9091
Epoch 154, 38 0.06551556375; 1000 [00071000 1110340000, 1.431575, loss=0.0527] PSNR=23.9941; SSIM=0.9190
Validation: 1000 [00071000 1110340000, 1.441575, loss=0.0427] PSNR=24.1047; SSIM=0.9289
Epoch 155, 38 0.06551556375; 1000 [00071000 1110340000, 1.451575, loss=0.0327] PSNR=24.2153; SSIM=0.7994
Validation: 1000 [00071000 1110340000, 1.461575, loss=0.0227] PSNR=24.3259; SSIM=0.8381
Epoch 156, 38 0.06551556375; 1000 [00071000 1110340000, 1.471575, loss=0.0127] PSNR=24.4365; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.481575, loss=0.0027] PSNR=24.5471; SSIM=0.8685
Epoch 157, 38 0.06551556375; 1000 [00071000 1110340000, 1.491575, loss=0.1364] PSNR=24.6578; SSIM=0.7985
Validation: 1000 [00071000 1110340000, 1.501575, loss=0.1264] PSNR=24.7684; SSIM=0.8183
Epoch 158, 38 0.06551556375; 1000 [00071000 1110340000, 1.511575, loss=0.1164] PSNR=24.8790; SSIM=0.8382
Validation: 1000 [00071000 1110340000, 1.521575, loss=0.1064] PSNR=24.9896; SSIM=0.7993
Epoch 159, 38 0.06551556375; 1000 [00071000 1110340000, 1.531575, loss=0.0964] PSNR=25.0992; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.541575, loss=0.0864] PSNR=25.2198; SSIM=0.8382
Epoch 160, 38 0.06551556375; 1000 [00071000 1110340000, 1.551575, loss=0.0764] PSNR=25.3304; SSIM=0.7715
Validation: 1000 [00071000 1110340000, 1.561575, loss=0.0664] PSNR=25.4410; SSIM=0.8382
Epoch 161, 38 0.06551556375; 1000 [00071000 1110340000, 1.571575, loss=0.0564] PSNR=25.5516; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.581575, loss=0.0464] PSNR=25.6622; SSIM=0.8382
Epoch 162, 38 0.06551556375; 1000 [00071000 1110340000, 1.591575, loss=0.0364] PSNR=25.7728; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.601575, loss=0.0264] PSNR=25.8834; SSIM=0.8382
Epoch 163, 38 0.06551556375; 1000 [00071000 1110340000, 1.611575, loss=0.0164] PSNR=25.9940; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.621575, loss=0.0064] PSNR=26.1046; SSIM=0.8382
Epoch 164, 38 0.06551556375; 1000 [00071000 1110340000, 1.631575, loss=0.0064] PSNR=26.2152; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.641575, loss=0.0064] PSNR=26.3258; SSIM=0.8382
Epoch 165, 38 0.06551556375; 1000 [00071000 1110340000, 1.651575, loss=0.0064] PSNR=26.4364; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.661575, loss=0.0064] PSNR=26.5470; SSIM=0.8382
Epoch 166, 38 0.06551556375; 1000 [00071000 1110340000, 1.671575, loss=0.0064] PSNR=26.6576; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.681575, loss=0.0064] PSNR=26.7682; SSIM=0.8382
Epoch 167, 38 0.06551556375; 1000 [00071000 1110340000, 1.691575, loss=0.0064] PSNR=26.8788; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.701575, loss=0.0064] PSNR=26.9894; SSIM=0.8382
Epoch 168, 38 0.06551556375; 1000 [00071000 1110340000, 1.711575, loss=0.0064] PSNR=27.0990; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.721575, loss=0.0064] PSNR=27.2196; SSIM=0.8382
Epoch 169, 38 0.06551556375; 1000 [00071000 1110340000, 1.731575, loss=0.0064] PSNR=27.3302; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.741575, loss=0.0064] PSNR=27.4408; SSIM=0.8382
Epoch 170, 38 0.06551556375; 1000 [00071000 1110340000, 1.751575, loss=0.0064] PSNR=27.5514; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.761575, loss=0.0064] PSNR=27.6620; SSIM=0.8382
Epoch 171, 38 0.06551556375; 1000 [00071000 1110340000, 1.771575, loss=0.0064] PSNR=27.7726; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.781575, loss=0.0064] PSNR=27.8832; SSIM=0.8382
Epoch 172, 38 0.06551556375; 1000 [00071000 1110340000, 1.791575, loss=0.0064] PSNR=27.9938; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.801575, loss=0.0064] PSNR=28.1044; SSIM=0.8382
Epoch 173, 38 0.06551556375; 1000 [00071000 1110340000, 1.811575, loss=0.0064] PSNR=28.2150; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.821575, loss=0.0064] PSNR=28.3256; SSIM=0.8382
Epoch 174, 38 0.06551556375; 1000 [00071000 1110340000, 1.831575, loss=0.0064] PSNR=28.4362; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.841575, loss=0.0064] PSNR=28.5468; SSIM=0.8382
Epoch 175, 38 0.06551556375; 1000 [00071000 1110340000, 1.851575, loss=0.0064] PSNR=28.6574; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.861575, loss=0.0064] PSNR=28.7680; SSIM=0.8382
Epoch 176, 38 0.06551556375; 1000 [00071000 1110340000, 1.871575, loss=0.0064] PSNR=28.8786; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.881575, loss=0.0064] PSNR=28.9892; SSIM=0.8382
Epoch 177, 38 0.06551556375; 1000 [00071000 1110340000, 1.891575, loss=0.0064] PSNR=29.0998; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.901575, loss=0.0064] PSNR=29.2104; SSIM=0.8382
Epoch 178, 38 0.06551556375; 1000 [00071000 1110340000, 1.911575, loss=0.0064] PSNR=29.3210; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.921575, loss=0.0064] PSNR=29.4316; SSIM=0.8382
Epoch 179, 38 0.06551556375; 1000 [00071000 1110340000, 1.931575, loss=0.0064] PSNR=29.5422; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.941575, loss=0.0064] PSNR=29.6528; SSIM=0.8382
Epoch 180, 38 0.06551556375; 1000 [00071000 1110340000, 1.951575, loss=0.0064] PSNR=29.7634; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.961575, loss=0.0064] PSNR=29.8740; SSIM=0.8382
Epoch 181, 38 0.06551556375; 1000 [00071000 1110340000, 1.971575, loss=0.0064] PSNR=29.9846; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 1.981575, loss=0.0064] PSNR=30.0952; SSIM=0.8382
Epoch 182, 38 0.06551556375; 1000 [00071000 1110340000, 1.991575, loss=0.0064] PSNR=30.2058; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.001575, loss=0.0064] PSNR=30.3164; SSIM=0.8382
Epoch 183, 38 0.06551556375; 1000 [00071000 1110340000, 2.011575, loss=0.0064] PSNR=30.4270; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.021575, loss=0.0064] PSNR=30.5376; SSIM=0.8382
Epoch 184, 38 0.06551556375; 1000 [00071000 1110340000, 2.031575, loss=0.0064] PSNR=30.6482; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.041575, loss=0.0064] PSNR=30.7588; SSIM=0.8382
Epoch 185, 38 0.06551556375; 1000 [00071000 1110340000, 2.051575, loss=0.0064] PSNR=30.8684; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.061575, loss=0.0064] PSNR=30.9790; SSIM=0.8382
Epoch 186, 38 0.06551556375; 1000 [00071000 1110340000, 2.071575, loss=0.0064] PSNR=31.0896; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.081575, loss=0.0064] PSNR=31.1992; SSIM=0.8382
Epoch 187, 38 0.06551556375; 1000 [00071000 1110340000, 2.091575, loss=0.0064] PSNR=31.3098; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.101575, loss=0.0064] PSNR=31.4194; SSIM=0.8382
Epoch 188, 38 0.06551556375; 1000 [00071000 1110340000, 2.111575, loss=0.0064] PSNR=31.5290; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.121575, loss=0.0064] PSNR=31.6396; SSIM=0.8382
Epoch 189, 38 0.06551556375; 1000 [00071000 1110340000, 2.131575, loss=0.0064] PSNR=31.7492; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.141575, loss=0.0064] PSNR=31.8598; SSIM=0.8382
Epoch 190, 38 0.06551556375; 1000 [00071000 1110340000, 2.151575, loss=0.0064] PSNR=31.9694; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.161575, loss=0.0064] PSNR=32.0790; SSIM=0.8382
Epoch 191, 38 0.06551556375; 1000 [00071000 1110340000, 2.171575, loss=0.0064] PSNR=32.1896; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.181575, loss=0.0064] PSNR=32.2992; SSIM=0.8382
Epoch 192, 38 0.06551556375; 1000 [00071000 1110340000, 2.191575, loss=0.0064] PSNR=32.4098; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.201575, loss=0.0064] PSNR=32.5194; SSIM=0.8382
Epoch 193, 38 0.06551556375; 1000 [00071000 1110340000, 2.211575, loss=0.0064] PSNR=32.6290; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.221575, loss=0.0064] PSNR=32.7396; SSIM=0.8382
Epoch 194, 38 0.06551556375; 1000 [00071000 1110340000, 2.231575, loss=0.0064] PSNR=32.8492; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.241575, loss=0.0064] PSNR=32.9598; SSIM=0.8382
Epoch 195, 38 0.06551556375; 1000 [00071000 1110340000, 2.251575, loss=0.0064] PSNR=33.0694; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.261575, loss=0.0064] PSNR=33.1790; SSIM=0.8382
Epoch 196, 38 0.06551556375; 1000 [00071000 1110340000, 2.271575, loss=0.0064] PSNR=33.2896; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.281575, loss=0.0064] PSNR=33.3992; SSIM=0.8382
Epoch 197, 38 0.06551556375; 1000 [00071000 1110340000, 2.291575, loss=0.0064] PSNR=33.5098; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.301575, loss=0.0064] PSNR=33.6194; SSIM=0.8382
Epoch 198, 38 0.06551556375; 1000 [00071000 1110340000, 2.311575, loss=0.0064] PSNR=33.7290; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.321575, loss=0.0064] PSNR=33.8396; SSIM=0.8382
Epoch 199, 38 0.06551556375; 1000 [00071000 1110340000, 2.331575, loss=0.0064] PSNR=33.9492; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.341575, loss=0.0064] PSNR=34.0598; SSIM=0.8382
Epoch 200, 38 0.06551556375; 1000 [00071000 1110340000, 2.351575, loss=0.0064] PSNR=34.1694; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.361575, loss=0.0064] PSNR=34.2790; SSIM=0.8382
Epoch 201, 38 0.06551556375; 1000 [00071000 1110340000, 2.371575, loss=0.0064] PSNR=34.3896; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.381575, loss=0.0064] PSNR=34.4992; SSIM=0.8382
Epoch 202, 38 0.06551556375; 1000 [00071000 1110340000, 2.391575, loss=0.0064] PSNR=34.6098; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.401575, loss=0.0064] PSNR=34.7194; SSIM=0.8382
Epoch 203, 38 0.06551556375; 1000 [00071000 1110340000, 2.411575, loss=0.0064] PSNR=34.8290; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.421575, loss=0.0064] PSNR=34.9396; SSIM=0.8382
Epoch 204, 38 0.06551556375; 1000 [00071000 1110340000, 2.431575, loss=0.0064] PSNR=35.0492; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.441575, loss=0.0064] PSNR=35.1598; SSIM=0.8382
Epoch 205, 38 0.06551556375; 1000 [00071000 1110340000, 2.451575, loss=0.0064] PSNR=35.2694; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.461575, loss=0.0064] PSNR=35.3790; SSIM=0.8382
Epoch 206, 38 0.06551556375; 1000 [00071000 1110340000, 2.471575, loss=0.0064] PSNR=35.4896; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.481575, loss=0.0064] PSNR=35.5992; SSIM=0.8382
Epoch 207, 38 0.06551556375; 1000 [00071000 1110340000, 2.491575, loss=0.0064] PSNR=35.7098; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.501575, loss=0.0064] PSNR=35.8194; SSIM=0.8382
Epoch 208, 38 0.06551556375; 1000 [00071000 1110340000, 2.511575, loss=0.0064] PSNR=35.9290; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.521575, loss=0.0064] PSNR=36.0396; SSIM=0.8382
Epoch 209, 38 0.06551556375; 1000 [00071000 1110340000, 2.531575, loss=0.0064] PSNR=36.1492; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.541575, loss=0.0064] PSNR=36.2598; SSIM=0.8382
Epoch 210, 38 0.06551556375; 1000 [00071000 1110340000, 2.551575, loss=0.0064] PSNR=36.3694; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.561575, loss=0.0064] PSNR=36.4790; SSIM=0.8382
Epoch 211, 38 0.06551556375; 1000 [00071000 1110340000, 2.571575, loss=0.0064] PSNR=36.5896; SSIM=0.7774
Validation: 1000 [00071000 1110340000, 2.581575, loss=0.0064] PSNR=36.6992; SSIM=0.8382
Epoch 212, 38 0.06551556375; 1000 [00071000 1110340000, 2.591575, loss=0.0064] PSNR=36.8098; SSIM=0.7774
Validation: 1000 [00071000 
```

```

5     dataset_dicts = []
6     files = [file for file in os.listdir(path) if
7             file.endswith(".json")]
8
9     for idx, json_filename in enumerate(files):
10        json_file = os.path.join(path, json_filename)
11        with open(json_file) as fptr:
12            img_annotations = json.load(fptr)
13
14        record = {}
15        image_filename = os.path.join(path,
16            img_annotations["imagePath"])
17        record["file_name"] = image_filename
18        record["image_id"] = idx
19
20        height, width = cv2.imread(image_filename).shape[:2]
21        record["width"], record["height"] = width,
22        height #720, 480
23
24        annotations = img_annotations["shapes"]
25        objs = []
26        for ann in annotations:
27            px = [a[0] for a in ann["points"]] # x-
28            coordinate (top-left and bottom-right corners)
29            py = [a[1] for a in ann["points"]] # y-
30            coordinate (top-left and bottom-right corners)
31
32            px = [px[0],px[1],px[0],px[1]]
33            py = [py[0],py[0],py[1],py[1]]
34
35            poly = [(x, y) for x, y in zip(px, py)]
36            poly = [p for x in poly for p in x]
37
38            obj = {"bbox": [np.min(px), np.min(py), np.
39            max(px), np.max(py)],
40            "bbox_mode": BoxMode.XYXY_ABS,
41            "segmentation": [poly],
42            "category_id": classes.index(ann['label']),
43            "iscrowd": 0}
44
45            objs.append(obj)
46
47            record["annotations"] = objs
48            dataset_dicts.append(record)
49
50    return dataset_dicts
51
52 def set_up_model(max_iter=500, save=None):
53     cfg = get_cfg()
54     cfg.MODEL.DEVICE = 'cpu'
55     cfg.merge_from_file(model_zoo.get_config_file("COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml"))
56     cfg.DATASETS.TRAIN = ("category_train",)
57     cfg.DATASETS.TEST = ()
58     cfg.DATA_LOADER.NUM_WORKERS = 2
59     cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml")
60     cfg.SOLVER.IMS_PER_BATCH = 2
61     cfg.SOLVER.BASE_LR = 0.00025
62     cfg.SOLVER.MAX_ITER = max_iter
63     cfg.MODEL.ROI_HEADS.NUM_CLASSES = 2
64
65     if(save):
66         with open(save, 'wb') as fptr:
67             pickle.dump(cfg, fptr, protocol=pickle.HIGHEST_PROTOCOL)
68
69     return cfg
70
71 def train_model(cfg):
72     os.makedirs(cfg.OUTPUT_DIR, exist_ok=True)
73     trainer = DefaultTrainer(cfg)
74
75     trainer.resume_or_load(resume=False)
76     trainer.train()
77
78 def semantic_segmentation(image, labels, masks):
79     blank = np.zeros(image.shape)
80     blank[:, :] = (139, 10, 80)
81     color = None
82     for label, mask in zip(labels, masks):
83         if(not label):
84             color = (255, 0, 0)
85         else:
86             color = (0, 0, 255)
87
88         for row in range(len(mask)):
89             for col in range(len(mask[row])):
90                 if(mask[row][col]):
91                     blank[row][col] = color
92
93     return blank
94
95 def prediction(cfg_path, image_path, metadata,
96     filename=None, bounding_box=False, thresh=0.3):
97     with open(cfg_path, 'rb') as fptr:
98         cfg = pickle.load(fptr)
99
100    cfg.MODEL.WEIGHTS = os.path.join(cfg.OUTPUT_DIR,
101        "model_final.pth")
102    cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = thresh
103
104    predictor = DefaultPredictor(cfg)
105    image = cv2.imread(image_path)
106    outputs = predictor(image)
107    visualizer(image, outputs, metadata, filename=
108        filename, bounding_box=bounding_box)

```

Listing 1. Detectron2 Source Code

D. Results

With a threshold of 0.3, the detectron2 network achieved an average precision of 95% with a training loss of 0.4077. Figures 3 - 6 are the outputs obtained from the detectron2 network.



Fig. 3. Detectron2 Output Sample 1

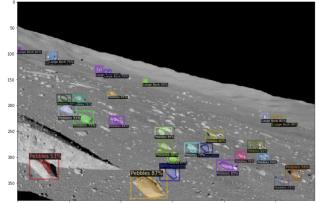


Fig. 4. Detectron2 Output Sample 2



Fig. 5. Detectron2 Output Sample 3



Fig. 6. Detectron2 Output Sample 4

IV. IMPLEMENTATION OF BEV NETWORK

A. Architecture of BeV Network

The paper proposes a neural network that processes all non-transformed images from a vehicle cameras as input.

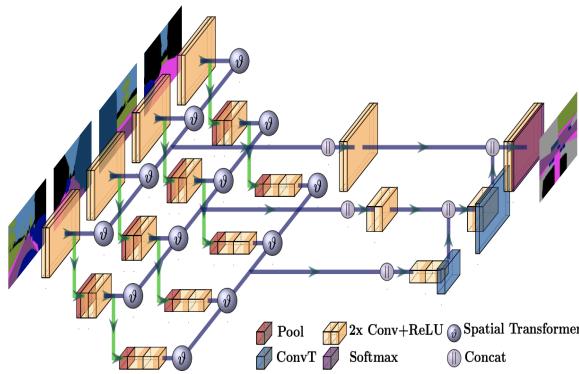


Fig. 7. Architecture of UnetXST [2]

It extracts features in the non-transformed camera views such that it is not subjected to the errors introduced by the IPM. To tackle the problem of spatial inconsistency, the paper integrates projective transformations into the network [2]. The architecture is designed such that it is capable of accepting multiple inputs for one output image. This is made possible by building on an existing network called *U-Net* [2].

The basic architecture as shown in figure 7 consists of a convolutional encoder and decoder path based on successive pooling and up-sampling, respectively [2]. In addition, high resolution features from the encoder side of the network are combined with up-sampled outputs on the decoder side via skip-connections on each scale [2]. Two extensions made to the *U-Net* network are:

- 1) The encoder path is separately replicated for each input image. For every scale, features from each input stream are concatenated and convoluted to build the skip connection to the single decoder path [2]
- 2) Before concatenating the input streams, *Spacial Transformer* units projectively transform the feature maps using the fixed homography as obtained by IPM [2]

These extensions are the reason for why the architecture is called *UnetXST* Network.

B. Training Datasets

The data used to train the *UnetXST* model to transform a front-facing image to a BeV perspective was created in the simulation environment Virtual Test Drive (VTD). In the simulation, the ego vehicle is equipped with a front-facing camera and the ground truth data is provided by a virtual drone camera centered above the ego vehicle. All virtual cameras produce both realistic and semantically segmented images [2].

C. Implementation

The table above shows the parameters that were used to train the *UnetXST* network. The parameters are identical to the author's implementation to make an attempt at replicating their results.

Parameters	Value
Epochs	100
Learning Rate	1e-4
Optimizer	Adam
Train Batch Size	2

TABLE III
THE PARAMETERS USED TO TRAIN UNETXST

D. Results

Figure 8 shows the results of my training for the last few epochs out of which these are the best results:

- Best Validation Loss: 0.1195
- Best Validation Categorical Accuracy: 0.9762
- Best Validation Mean IOU ³: 0.8800

Skip 1st	1
Skip 2nd	1
Skip 3rd	1
Skip 4th	1
Skip 5th	1
Skip 6th	1
Skip 7th	1
Skip 8th	1
Skip 9th	1
Skip 10th	1
Skip 11th	1
Skip 12th	1
Skip 13th	1
Skip 14th	1
Skip 15th	1
Skip 16th	1
Skip 17th	1
Skip 18th	1
Skip 19th	1
Skip 20th	1
Skip 21st	1
Skip 22nd	1
Skip 23rd	1
Skip 24th	1
Skip 25th	1
Skip 26th	1
Skip 27th	1
Skip 28th	1
Skip 29th	1
Skip 30th	1
Skip 31st	1
Skip 32nd	1
Skip 33rd	1
Skip 34th	1
Skip 35th	1
Skip 36th	1
Skip 37th	1
Skip 38th	1
Skip 39th	1
Skip 40th	1
Skip 41st	1
Skip 42nd	1
Skip 43rd	1
Skip 44th	1
Skip 45th	1
Skip 46th	1
Skip 47th	1
Skip 48th	1
Skip 49th	1
Skip 50th	1
Skip 51st	1
Skip 52nd	1
Skip 53rd	1
Skip 54th	1
Skip 55th	1
Skip 56th	1
Skip 57th	1
Skip 58th	1
Skip 59th	1
Skip 60th	1
Skip 61st	1
Skip 62nd	1
Skip 63rd	1
Skip 64th	1
Skip 65th	1
Skip 66th	1
Skip 67th	1
Skip 68th	1
Skip 69th	1
Skip 70th	1
Skip 71st	1
Skip 72nd	1
Skip 73rd	1
Skip 74th	1
Skip 75th	1
Skip 76th	1
Skip 77th	1
Skip 78th	1
Skip 79th	1
Skip 80th	1
Skip 81st	1
Skip 82nd	1
Skip 83rd	1
Skip 84th	1
Skip 85th	1
Skip 86th	1
Skip 87th	1
Skip 88th	1
Skip 89th	1
Skip 90th	1
Skip 91st	1
Skip 92nd	1
Skip 93rd	1
Skip 94th	1
Skip 95th	1
Skip 96th	1
Skip 97th	1
Skip 98th	1
Skip 99th	1
Skip 100th	1

Fig. 8. UnetXST Training

V. CREATION OF THE PIPELINE

To combine the deblur, detectron2, and BeV networks and deploy them onto the lunar rover the club is currently building, I use an open source framework for robotic applications called ROS to get access to a continuous stream of images from the camera on the rover as my inputs to my networks.

A. Source Code

Below is the source code for the ROS deployment of the pipeline.

```

1 import rospy
2 import numpy as np
3 from sensor_msgs.msg import Image
4 import cv2
5 from deblurring.predict import Predictor
6 from detectron.detectron import prediction
7 from BeV.predict import predict
8 from cv_bridge import CvBridge
9
10 import os
11 os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
12
13 # Global Variables
14 bridge = CvBridge()

```

³Intersection or overlap between ground truth and result of network

```

15 publisher = rospy.Publisher("/cv_pipeline_result/", 72 image = cv2.imread("/home/parallels/catkin_ws/
16     data_class=Image, queue_size=10) 73     src/cv/scripts/render0909.png")
17 result = None 74     result = cv_pipeline(image)
18 def deblur(image): 75     result = bridge.cv2_to_imgmsg(result, encoding=
19     weights_path = r"/home/parallels/catkin_ws/src/ 76     "passthrough")
20     cv/scripts/deblurring/best_fpn.h5" 77     publisher.publish(result)
21     predictor = Predictor(weights_path=weights_path 78     rospy.loginfo("Image Published")
22     ) 79
23     image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB) 80 if __name__ == "__main__":
24 81     rospy.init_node("ros_wrapper", anonymous=True)
25     deblurred = predictor(image, mask=None) 82     rospy.loginfo("Node has been initialized")
26     deblurred = cv2.cvtColor(deblurred, cv2. 83
27     COLOR_RGB2BGR) 84     rospy.Subscriber("/camera/color/image_raw",
28 85         data_class=Image, callback=callback)
29     return deblurred 86     rospy.loginfo("Node started") # simply keeps
30 87     python from exiting until this node is stopped
31 def segmentation(image): 88
32     cfg_path = r"/home/parallels/catkin_ws/src/cv/ 89     rospy.spin()
33     scripts/detectron/cfg_model.pickle" 90
34     meta_data = r"/home/parallels/catkin_ws/src/cv/ 91
35     scripts/detectron/output/ 92
36     category_test_coco_format.json" 93
37     model_weights_path = r"/home/parallels/ 94
38     catkin_ws/src/cv/scripts/detectron/output/ 95
39     model_final.pth" 96
40
41     segmented = prediction(cfg_path, 97
42     model_weights_path, image, meta_data, thresh= 98
43     0.3) 99
44
45     return segmented 100
46
47 def birds_eye_view(image): 101
48     image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB) 102
49     config_file_path = r"/home/parallels/catkin_ws/ 103
50     src/cv/scripts/BeV/config.2_F.unetxst.yml" 104
51     model_weight_path = r"/home/parallels/catkin_ws/ 105
52     /src/cv/scripts/BeV/best_weights.hdf5" 106
53
54     BeV_transformed = predict(image, 107
55     config_file_path, model_weight_path) 108
56     return BeV_transformed 109
57
58 def cv_pipeline(image, blur=False): 110
59     rospy.loginfo("Image is loaded") 111
60     cv2.imwrite("original.png", image) 112
61
62     if(blur): 113
63         # To purposefully blur an image for testing 114
64         purposes 115
65         image = cv2.GaussianBlur(image, ksize=(3,3), 116
66         sigmaX=1.3) 117
67         rospy.loginfo("Image is blurred") 118
68         cv2.imwrite("Blurred.png", image) 119
69
70         deblurred = deblur(image) 120
71         rospy.loginfo("Image is deblurred") 121
72         cv2.imwrite("deblurred.png", deblurred) 122
73
74         segmented = segmentation(deblurred) 123
75         rospy.loginfo("Image is segmented") 124
76         cv2.imwrite("segmented.png", segmented) 125
77
78         BeV_transformed = birds_eye_view(segmented) 126
79         rospy.loginfo("Image transformed to BeV") 127
80         cv2.imwrite("BeV.png", BeV_transformed) 128
81
82         return BeV_transformed 129
83
84 def callback(message): 130
85     image = bridge.imgmsg_to_cv2(message, 131
86     desired_encoding='bgr8') # shape: 480 x 640 x 3 132

```

Listing 2. The Pipeline

B. Results

By passing a blurred image of the lunar surface through the pipeline, the results are as shown in figure 9

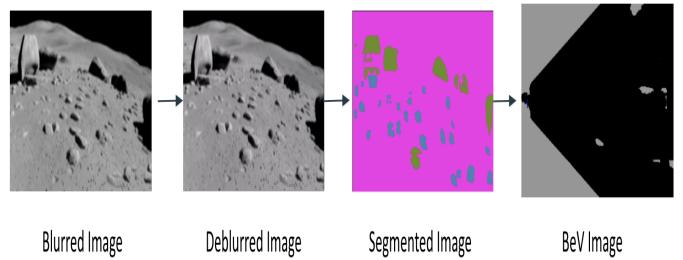


Fig. 9. Output of the Pipeline

VI. IMPLEMENTATION OF RGB-DEPTH RECONSTRUCTION

The last section of my senior design was to implement RGB-Depth reconstruction in order to provide the stakeholders of the option of creating a custom dataset to train the pipeline on.

A. Camera Calibration

Before I start implementing RGB-Depth reconstruction, it was vital to know what the **extrinsic** and **intrinsic** parameters of the camera is. The camera I used for this senior design is the one preferred by the club: Intel Realsense D455 camera.

The extrinsic and intrinsic parameters of a camera are transformation matrices that convert points from one coordinate system to the other [8]. The extrinsic matrix is a transformation matrix from the world coordinate system to the camera coordinate system, while the intrinsic matrix is a transformation matrix that converts points from the camera coordinate system to the pixel coordinate system [8].

The extrinsic and intrinsic parameters of the camera are obtained through Zhang's Algorithm as shown in a snippet of my source code below

```

1  ### Global Variables
2 bridge = CvBridge()
3 CHECKERBOARD = (6, 9) # Dimensions of the
4 criteria = (cv2.TERM_CRITERIA_EPS + cv2.
    TERM_CRITERIA_MAX_ITER, 30, 0.001) #
        Termination Criteria
5
6 def get_corners(image, gray, objp, display=False):
7     # Find the chessboard corners
8     # If desired number of corners are found in the
9     # image then ret = true
10    # cv2.CALIB_CB_ADAPTIVE_THRESH = Use adaptive
11    # thresholding to convert the image to black and
12    # white, rather than a fixed threshold level (
13    # computed from the average image brightness)
14    # cv2.CALIB_CB_FAST_CHECK = Run a fast check on
15    # the image that looks for chessboard corners,
16    # and shortcut the call if none is found. This
17    # can drastically speed up the call in the
18    # degenerate condition when no chessboard is
19    # observed.
20    # cv2.CALIB_CB_NORMALIZE_IMAGE = Normalize the
21    # image gamma with equalizeHist before applying
22    # fixed or adaptive thresholding.
23    ret, corners = cv2.findChessboardCorners(gray,
24        patternSize=CHECKERBOARD, flags=cv2.
25        CALIB_CB_ADAPTIVE_THRESH + cv2.
26        CALIB_CB_FAST_CHECK + cv2.
27        CALIB_CB_NORMALIZE_IMAGE)
28
29    # If desired number of corners are detected, we
30    # refine the pixel coordinates and display them
31    # on the checkerboard
32    if(ret):
33        corners2 = cv2.cornerSubPix(gray, corners=
34            corners, winSize=(11, 11), zeroZone=(-1, -1),
35            criteria=criteria) # refining pixel coordinates
36            for given 2d points
37            if(display):
38                display_corners(image, corners2, ret)
39
40            return objp, corners2
41
42    else:
43        return None, None
44
45 def get_intrinsic_extrinsic_parameters(gray,
46     obj_points, img_points):
47     """
48     Performing camera calibration by
49     passing the value of known 3D points (objpoints
50     )
51     and corresponding pixel coordinates of the
52     detected corners (imgpoints)
53     """
54
55     # K = Intrinsic Matrix
56     # dist = Lens distortion coefficients. These
57     # coefficients will be explained in a future post
58     .
59     # R = Rotation Rodrigues Vector
60     # t = Translation Vector
61
62     ret, K, dist, rvecs, tvecs = cv2.
63     calibrateCamera(obj_points, img_points, gray.
64     shape[::-1], None, None)
65     R, jacobian = cv2.Rodrigues(rvecs[0])
66     t = tvecs[0]
67
68     return K, R, t

```

B. Implementation

Once we get the camera parameters, I implement RGB-Depth reconstruction by creating a pointcloud by storing the depth value at each pixel representation of the 2d RGB image obtained from the camera attached on the lunar rover via ROS as shown in the following snippet of the code

```

1 def get_camera_parameters(self):
2     parameters = np.load(r"/home/parallels/
3         catkin_ws/src/cv/scripts/RGBd/parameters.npz")
4     self.K, self.R, self.t = np.array(parameters["K"])
5     , np.array(parameters["R"]),
6     np.array(parameters["t"]))
7
8     self.K = np.column_stack((self.K, np.array
9     ([0, 0, 0])))
10    self.K = np.row_stack((self.K, np.array
11    ([0, 0, 0, 1])))
12
13    self.Rt = np.column_stack((self.R, self.t))
14    self.Rt = np.row_stack((self.Rt, np.array
15    ([0, 0, 0, 1])))
16
17 def display_point_cloud(self, display=False):
18     self.get_camera_parameters()
19     fx, cx, fy, cy = self.K[0, 0], self.K[0, 2], self
20     .K[1, 1], self.K[1, 2]
21
22     self.rgbd_image = o3d.geometry.RGBDImage.
23     create_from_color_and_depth(self.geometry_rgb,
24     self.geometry_depth, convert_rgb_to_intensity=
25     False)
26     height, width = self.rgb_image.shape[:2]
27     point_cloud = o3d.geometry.PointCloud.
28     create_from_rgbd_image(self.rgbd_image, o3d.
29     camera.PinholeCameraIntrinsic(width=width,
30     height=height, fx=fx, fy=fy, cx=cx, cy=cy))
31
32     # Flip it, otherwise the pointcloud will be
33     # upside down
34     point_cloud.transform([[1, 0, 0, 0], [0, -1, 0,
35     0], [0, 0, -1, 0], [0, 0, 0, 1]])
36
37     if(display):
38         o3d.visualization.draw_geometries([
39             point_cloud])
40
41     return point cloud

```

Listing 4. RGB Depth Reconstruction

C. Results

Figure 10 illustrates the output of the RGB-Depth reconstruction. The result manages to capture the 3D essence from just an RGB image captured by the rover.



Listing 3. Camera Parameters

Fig. 10. Sample of RGB-Depth Result

VII. CONCLUSION

Through this senior design, I was able to successfully produce a prototype to transform a front-facing image to its BeV perspective which is essential when creating a localization map of the rover with respect to its surroundings. By successfully creating a prototype, I was able to assist the Lunabotics team in getting one step closer to making the rover autonomous which is a very significant metric in the NASA competition.

Some next steps for future teams would be to continue making improvements upon this prototype by gaining access to more datasets to train the networks on so that their validation metric scores could increase, by identifying craters on the surface of the moon during semantic segmentation, and finally using the outputs of the pipeline to create a localization system using SLAM.

VIII. ACKNOWLEDGEMENTS

I would like to thank Professor Nicole Ramirez and Graduate Teaching Assistant Prabhpreet Dhir for without whom I would not have been able to complete this senior design project.

REFERENCES

- [1] Orest Kupyn, Tetiana Martyniuk, Junru Wu, Zhangyang Wang (2019) *DeblurGAN-v2: Deblurring (Orders-of-Magnitude) Faster and Better*
- [2] Lennart Reiher, Bastian Lampe, Lutz Eckstein (2020) *A Sim2Real Deep Learning Approach for the Transformation of Images from Multiple Vehicle-Mounted Cameras to a Semantically Segmented Image in Bird's Eye View*
- [3] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke (2016) *Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning*
- [4] Christopher Thomas (2020) *Deep learning image enhancement insights on loss function engineering*
- [5] Pranjal Datta (2020) *All about Structural Similarity Index (SSIM): Theory + Code in PyTorch*
- [6] Kaiming He, Georgia Gkioxari, Piotr Dollar, Ross Girshick (2018) *Mask R-CNN*
- [7] Romain Pessia (2018) *Artificial Lunar Landscape Dataset*
- [8] Aqeel Anwar (02/28) *What are Intrinsic and Extrinsic Camera Parameters in Computer Vision?*