Segment features around residential buildings in UAV images

Kaggle Competition

Leopold Hebert-Stevens
Adrien Loizeau

Maria Vakalopoulou

January 20, 2023

Contents

1 Introduction

In this report, we describe the implementation and evaluation of our UNet deep learning model for the task of automatic image segmentation. The goal is to segment UAV images of Houston, Texas after Hurricane Harvey, in order to assess the damages on residential and public properties. The images have 25 classes of segments, including various types of structures, vegetation, and vehicles.

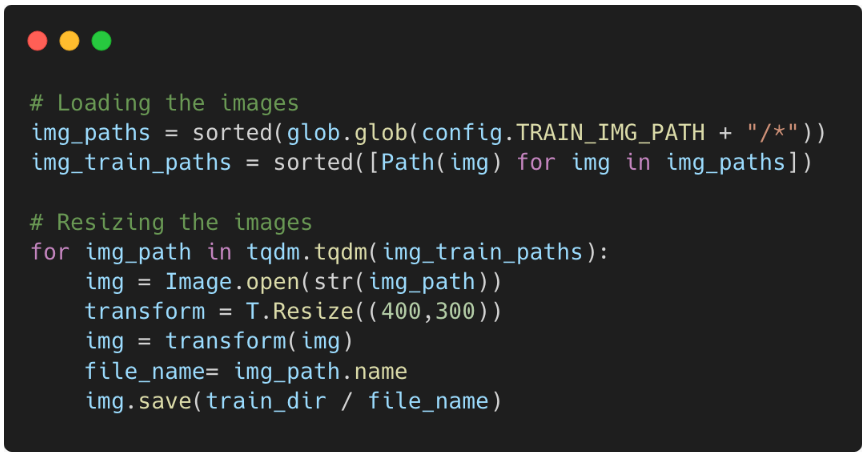
In the following sections of this report, we will describe the data preprocessing, model selection and training, evaluation, and results of our approach to this task. We will also discuss the challenges and limitations of our approach and suggest potential directions for future work.

2 Improving the speed of iteration

2.1 Resizing images

One of the main issues we had when we started working on this project was the time the Dataloader were taking to load the datasets. The vanilla code was taking more than 120 seconds on our computers for this task. The reason was the size of the dataset which had hundred of 4000 per 3000 images, resulting in a 1.8 GB dataset.

Our main idea to tackle this problem was to reduce by a factor 10 the size of all the images. To do so we imported the whole dataset and simply resized the images one by one using Pytorch's functional methods. An example of the code is displayed in Figure 1. We applied this method to the train-set, the test-set and also to the masks. Below (3) is an example of a resizing.



```python
# Loading the images
img_paths = sorted(glob.glob(config.TRAIN_IMG_PATH + "/*"))
img_train_paths = sorted([Path(img) for img in img_paths])

# Resizing the images
for img_path in tqdm.tqdm(img_train_paths):
    img = Image.open(str(img_path))
    transform = T.Resize((400,300))
    img = transform(img)
    file_name= img_path.name
    img.save(train_dir / file_name)
```

Figure 1: Resizing code example

Once all the images were reshaped, we decided to upload this new dataset to kaggle. This helped us in the long run to save some precious minutes. The dataset is available [here]. This new dataset delivers the same performances in training with a loading time of only 5 seconds.

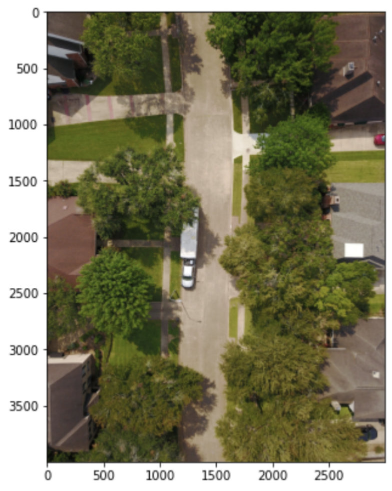


Figure 2: Original image of shape (4000,3000)

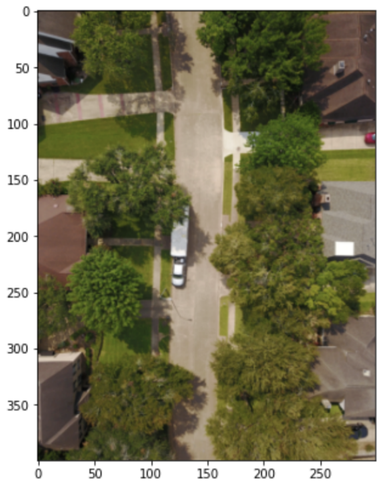


Figure 3: Image resized to shape (400,300)

2.2 New evaluation metrics

One issue with the original baseline code was the metric used to evaluate the model's performance, which was simply the accuracy. This may not be the most appropriate metric for this task, as other factors such as the overlap between the predicted and ground truth segments should also be considered. As the one used for kaggle leaderboard was the Dice metric, we decided to implement it.

As a quick reminder, the Dice coefficient is a metric used to evaluate the performance of image segmentation models. It measures the overlap between the predicted segmentation (A) and the ground truth segmentation (B). The Dice coefficient ranges from 0 to 1, with a value of 1 indicating a perfect overlap. It is particularly useful for imbalanced datasets, where other metrics such as accuracy may be misleading.

$$\text{Dice} = \frac{2 * (A \cap B)}{A + B}$$

To implement it, we used the TorchMetrics library's Dice Metric which allowed us to simply add it.

2.3 Use of GPUs

We decided to use GPUs for two main reasons:

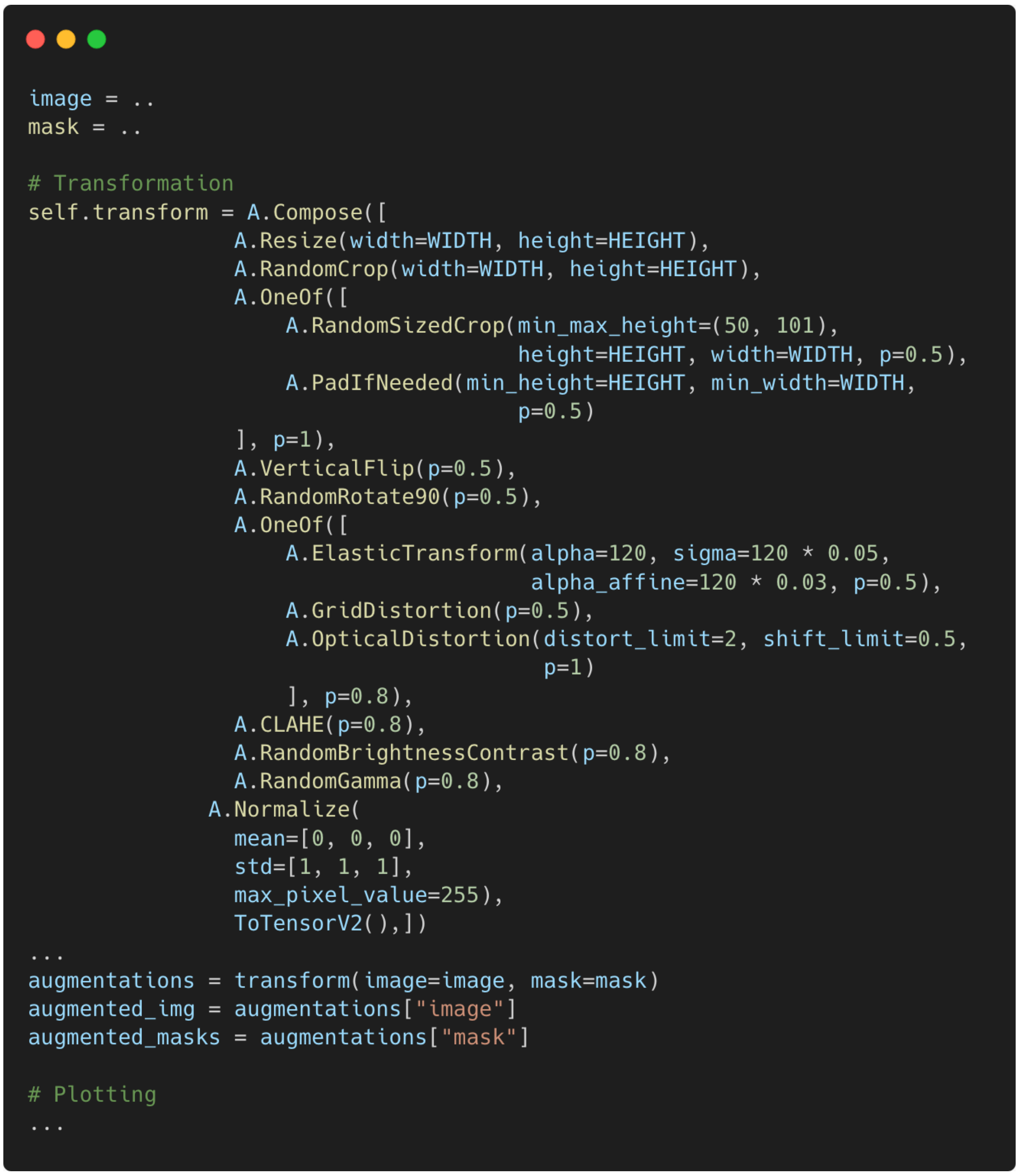
- Speed: GPUs are much faster than out computer CPUs at performing the matrix operations that are fundamental when manipulating images such as image segmentation.
- Memory: GPUs have a lot of memory, which is important when training deep learning models that have insane amounts of parameters. This allows us to train larger models and use larger batch sizes, which improves the accuracy of the model.

Overall, the use of GPUs significantly accelerates the training of our deep learning models, which makes it possible to train larger and more accurate models in a shorter amount of time.

3 Improving performance

3.1 Data augmentation and transformations

The first modification we did to improve the model's performances was apply data augmentation. This part was very challenging as we needed to apply the same transformations to both an image and its corresponding mask. To overcome this issue we used the Albumentations library, a Python library for fast and flexible image augmentation which provides a wide range of image transformation functions, including resizing, cropping, rotating, flipping, and color adjustments. The following code 4 is the one we implemented:



```
image = ..
mask = ..

# Transformation
self.transform = A.Compose([
                A.Resize(width=WIDTH, height=HEIGHT),
                A.RandomCrop(width=WIDTH, height=HEIGHT),
                A.OneOf([
                    A.RandomSizedCrop(min_max_height=(50, 101),
                                      height=HEIGHT, width=WIDTH, p=0.5),
                    A.PadIfNeeded(min_height=HEIGHT, min_width=WIDTH,
                                      p=0.5)
                ], p=1),
                A.VerticalFlip(p=0.5),
                A.RandomRotate90(p=0.5),
                A.OneOf([
                    A.ElasticTransform(alpha=120, sigma=120 * 0.05,
                                       alpha_affine=120 * 0.03, p=0.5),
                    A.GridDistortion(p=0.5),
                    A.OpticalDistortion(distort_limit=2, shift_limit=0.5,
                                        p=1)
                    ], p=0.8),
                A.CLAHE(p=0.8),
                A.RandomBrightnessContrast(p=0.8),
                A.RandomGamma(p=0.8),
              A.Normalize(
                mean=[0, 0, 0],
                std=[1, 1, 1],
                max_pixel_value=255),
                ToTensorV2(),])
...
augmentations = transform(image=image, mask=mask)
augmented_img = augmentations["image"]
augmented_masks = augmentations["mask"]

# Plotting
...
```

Figure 4: Transformation code

We applied a series of transformations to an image in order to augment it. These transformations include:

- A.Resize: This transformation resizes the image to the specified width and height.
- A.RandomCrop: This transformation randomly crops the image to the specified width and height.
- A.OneOf: This transformation applies either a RandomSizedCrop or a PadIfNeeded transformation, with equal probability. The RandomSizedCrop transformation randomly crops the image to a size within a specified range, while the PadIfNeeded transformation pads the image with zeros if it is smaller than the specified size. These transformations are useful for adjusting the aspect ratio of the image or for ensuring that it has a minimum size.
- A.VerticalFlip: This transformation flips the image vertically with probability 0.5.
- A.RandomRotate90: This transformation rotates the image by a random multiple of 90 degrees with probability 0.5.
- A.OneOf: This transformation applies either an ElasticTransform, GridDistortion, or OpticalDistortion transformation with probability 0.8. The ElasticTransform transformation applies a non-rigid image deformation, the GridDistortion transformation distorts the image by moving pixels along a grid, and the OpticalDistortion transformation distorts the image by shifting pixels along a sine curve. These transformations can be useful for training models to be robust to minor deformations or distortions in the image.
- A.CLAHE: This transformation applies contrast limited adaptive histogram equalization to the image with probability 0.8. This can be useful for enhancing the contrast in images that have low contrast or for removing color casts. This transformation is only applied to the image and not the mask.
- A.RandomBrightnessContrast: This transformation adjusts the brightness and contrast of the image randomly with probability 0.8. This is for training models to be robust to variations in lighting conditions. This transformation and the following ones have been only applied to the image.
- A.RandomGamma: This transformation adjusts the gamma value of the image randomly with probability 0.8. This is useful for training models to be robust to variations in the overall intensity of the image.
- A.Normalize: This transformation normalizes the image by subtracting the mean and dividing by the standard deviation, and scales the resulting image so that the maximum pixel value is 255. This is useful for scaling the image to a specific range or for standardizing the overall intensity of the image.
- ToTensorV2: This transformation converts the image to a tensor. This is often required when training machine learning models, as most models expect input data to be in tensor format.

These transformations are composed using the A.Compose function, which applies all the transformations in the specified order. The resulting transform object can then be used to augment images as needed. Here are a few sample of the images generated thanks to this data-augmentation:

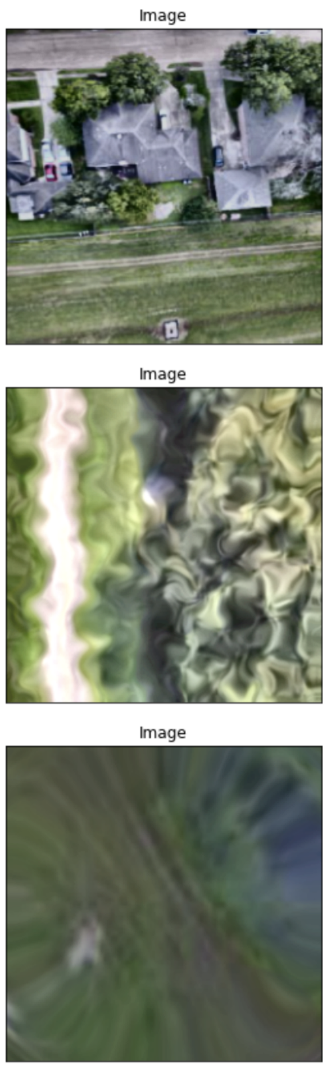


Figure 5: Transformations applied to the image

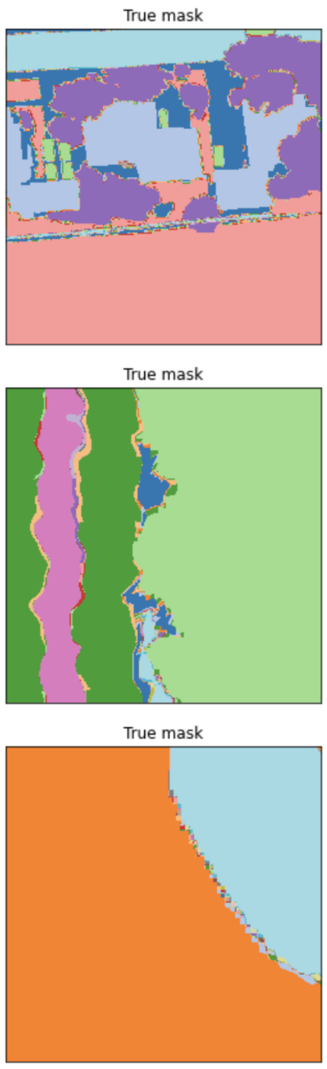


Figure 6: Transformations applied to the mask

3.2 New architecture: UNet

3.2.1 Description of the architecture

The U-Net model consists of an encoder part and a decoder part. The encoder part of the model consists of a series of convolutional layers that downsample the input image by reducing its spatial resolution through max pooling. The decoder part of the model consists of a series of transposed convolutional layers (also known as deconvolutional layers) that upsample the encoded feature maps from the encoder part to reconstruct the original spatial resolution of the input image. The U-Net model also includes skip connections, which concatenate the feature maps from the encoder and decoder parts at corresponding scales and help to improve the model's performance.

In this implementation, the encoder part of the model consists of four "BasicBlock" modules, each of which applies two convolutional layers and a max pooling operation to the input image to reduce its size by a factor of 2. The decoder part of the model consists of eight "BasicBlock" modules that use transposed convolutional layers to upsample the feature maps from the encoder part, as well as a series of max pooling operations to interpolate the upsampled feature maps. Finally, the model includes a bottleneck layer at the center of the U-shaped architecture, which helps to reduce the number of channels in the feature maps and prevent overfitting.

The model takes a 3-channel image as input and produces an output with the same spatial dimensions as the input, but with a different number of channels, specified by the "out_channels" parameter in the constructor. Each channel of the output corresponds to a different class or feature that the model is trying to segment in the input image. The model is trained using a supervised learning approach, where it is provided with labeled examples of input images and their corresponding output masks, and it adjusts the weights of its layers to minimize the difference between the predicted output masks and the ground truth masks.

3.2.2 Advantages and disadvantages

This model was inspired by the baseline model, and the use of basic blocks was retained. However, skip connections were implemented at each down and up sample step, giving the model a way better accuracy than the baseline. But this increase in performance also brings disadvantages. Here are the pros and cons of this architecture.

Shortcut connections allow information to flow directly from the encoder to the decoder, which improves the model's ability to retain spatial information and make more accurate predictions. They also improve the stability of the training process by providing an alternative path for the gradients to flow through the network. It makes the model less sensitive to the initialisation of the weights and can help prevent vanishing gradients. Finally, skip connections also improve the model's performance by allowing it to use low-level features that would otherwise be lost as the information flows through the network. These advantages of skip connections are especially useful in our situation because it is important to retain spatial information and details about small objects in the image.

On the other hand, these are the disadvantages of using skip connections in our neural network architecture: First, they increase the memory usage of the model because they require the model to store and process the additional information being passed through the network. Due to the large amount of data processed in the model, Google Colab ran out of memory numerous times. Secondly, shortcut connections also increase the computational cost of the model because they require additional matrix operations to be performed. Finally, we realised that when the model was trained with a lot of epochs, the model would overfit. This is because skip connections allow the model to use more information and make it easier for the model to memorise the training data.

3.3 Model tuning

The model was tested on different hyperparameters, such as the learning rate, the loss criterion, the optimizer used, and the number of epochs. We used WandB to compare the performance of your models to see how different hyperparameters affect our model's performance. After comparing the loss and accuracy plots for different hyperparameters, we found that the model was best at 50 epochs, with a learning rate of 0.0001, using the CrossEntropyLoss, and the Adam optimizer.

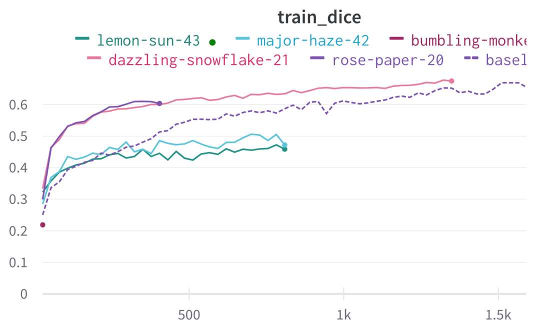


Figure 7: Train Dice evolution on Wandb

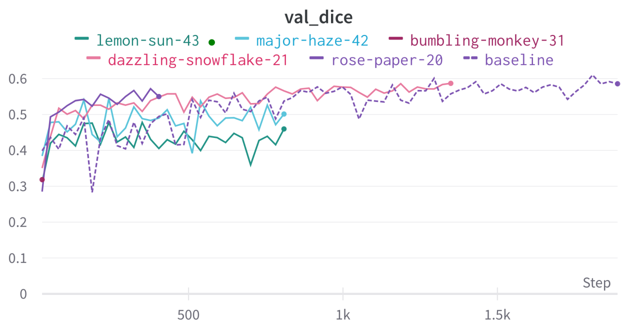


Figure 8: Validation Dice evolution on Wandb

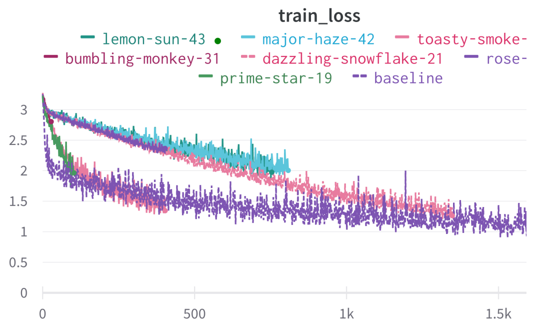


Figure 9: Train loss evolution on Wandb

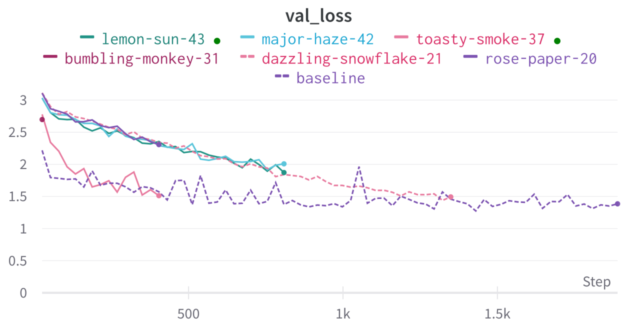


Figure 10: Validation loss evolution on Wandb

4 Conclusion

In this project, we aimed to improve the speed and performance of a baseline deep learning model for image segmentation. To improve the speed of the data loading process, we reduced the size of the images by a factor of 10, which significantly reduced the loading time. We also implemented the Dice coefficient as a new evaluation metric, which is more suitable for imbalanced datasets such as the one used in this project. To further improve performance, we used GPUs to accelerate training and applied data augmentation and transformations using the albumentation library. We also implemented a U-Net architecture, which proved to be effective in improving the model's performance. Finally, we experimented with different hyperparameter settings and found that using a learning rate of 0.005 and a batch size of 8 gave the best results. Overall, these changes allowed us to improve the speed and performance of the model, leading to better results on the image segmentation task.

Nevertheless, even though our model performs very well, we could have implemented other features to increase its performance. With more time, we could have worked on a better train validation split, fine-tuned the model, used transfer learning, or even implemented the SLIC method from our computer vision class. The SLIC method could have been a valuable tool for improving the performance of our deep learning image segmentation algorithm by providing a more accurate and consistent set of seeds for the clustering process and reducing the complexity of the input image.