

Image Segmentation with U-Net and Transfer Learning

Wuyeh Jobe

Carnegie Mellon University Africa
Kigali Innovation City, Kigali, Rwanda
jwuyeh@andrew.cmu.edu

Abstract

Image segmentation has wide-ranging applications in computer vision. From scene understanding in autonomous driving to medical imaging, down to photo editing (use cases that have gained remarkable traction over the years), image segmentation is one of the most significant aspects of computer vision. Yet, it is a daunting challenge primarily because of the dynamism of images. Thus, there is not a one-size-fits-all algorithm for image segmentation. Nevertheless, the increase performance of computers heralded new ways of tackling image segmentation, such as deep learning, which have produced remarkable results. U-Net is a convolutional neural network that was developed for medical imaging. It has become popular for other tasks because of the possibility to modify it and train with less training data for better results over traditional image segmentation techniques. In this paper, we explain the process we went through to train the U-Net model on new training samples, and the techniques used to improve the accuracy results.

1. Introduction

The journey to complete this assignment began with an analysis of technologies available to carry out image segmentation. We looked at Simple Interactive Object Extraction (SIOX). This algorithm is relatively fast and noise-robust, making it plausible for even video segmentation. Yet it requires so much information about foreground and background to be able to do the segmentation accurately. Additionally, as noted here[1], the algorithm assumes a normal daylight scene and segmentation fail for images veering off from this assumption.

We also looked at the GrabCut segmentation algorithm. Before deep learning techniques, GrabCut is one of the segmentation algorithms that works quite well for a wide range of images. It works by accepting an input image with either a bounding box signifying the region of interest or a mask that approximates the segmentation [2]. We try to use

the implementation in OpenCV to see how it performs on the input image quickly. For this implementation, we specified the number of iterations to be done on the images. Due to this and because the images are diverse, GrabCut performed really well for some images (even better than U-Net results) but not great on others as shown later.

The final technique tried is the U-Net semantic segmentation. Semantic segmentation techniques are thought of as classification problems where you predict the class of each pixel in the image. The U-Net technique is the one that gave robust results across all the images and hence the final solution for this assignment.

2. Using U-Net in Tensorflow for image segmentation

2.1. Loading the dataset

While Tensorflow has documentation that load data from a directory from the computer, the approach shown was not suitable for image segmentation. We looked at many tutorials online and adapted the code to get it ready for training the model. This code for setting up the data as a Tensorflow dataset was largely adapted from here [3].

Firstly, we get the absolute path to all the images and divide them in train, test and validation split using scikit-learn. Secondly, we read the images (both input images and masks); resize them (because the U-Net architecture like most neural networks requires fixed-size images as inputs); and normalize them. Once this is done, we parse the data to create Tensorflow slices. Finally, to configure dataset for performance, we used buffered prefetching to yield data from disk without I/O blocking. The dataset is ready for training, testing and validation.

```
# Configure for performance
train_dataset = train.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE).repeat()
train_dataset = train_dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
validation_dataset = val.batch(BATCH_SIZE)
test_dataset = val.batch(BATCH_SIZE)
```

Figure 1: Configuring dataset for performance.

2.2. Setting up the model

Once we have the dataset in shape, we set up the model for training. The idea for setting up the model was drawn from this Tensorflow tutorial [4].

Firstly, for transfer learning, we use the MobileNet V2 model to create a base model. This model, developed by Google, is pre-trained on the ImageNet dataset, which consists of over 1 million images and 1000 classes. Next, we extract the layers from this model, except for the top classification layer, which is usually not very useful. Next, we create the feature extraction model from these layers and freeze the convolutional base before compiling and training the model, because we do not want the weights from the given layers being updated during training. After this, we build our decoder or upsampler. Once this is done, we build the U-Net model, as shown below.

```
def unet_model(output_channels):
    inputs = tf.keras.layers.Input(shape=[img_width, img_height, 3])
    x = inputs

    # Downsampling through the model
    skips = down_stack(x)
    x = skips[-1]
    skips = reversed(skips[:-1])

    # Upsampling and establishing the skip connections
    for up, skip in zip(up_stack, skips):
        x = up(x)
        concat = tf.keras.layers.Concatenate()
        x = concat([x, skip])

    # This is the last layer of the model
    last = tf.keras.layers.Conv2DTranspose(
        output_channels, 3, strides=2,
        padding='same') #64x64 -> 256x256

    x = last(x)

    return tf.keras.Model(inputs=inputs, outputs=x)

model = unet_model(OUTPUT_CHANNELS)
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Figure 2: Building the U-Net model and compiling.

As you may have noted, we resized the input layer to accept a specific image size, which is the same the resized value when preprocessing the data. We build the U-Net model by calling this function and pass it the number of channels. Also, noticed that the *SparseCategoricalCrossentropy* is used as a loss function because, as noted earlier, semantic segmentation is regarded as a classification task where we predict each pixel. We are now ready for training.

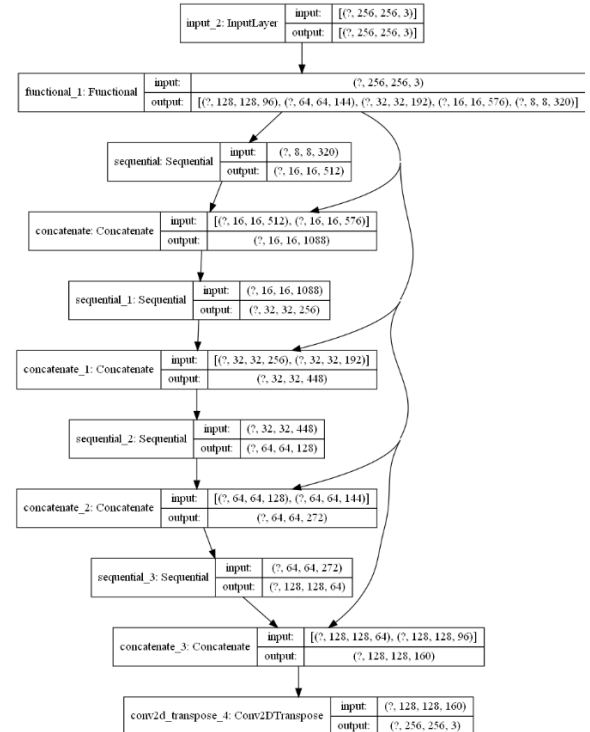


Figure 3: U-Net Model Architecture

2.3. Training

Once the model is set up, we can begin training. We first load the data, as explained earlier and then passed it in the model. Initially, the model was trained on with 65 samples. This represents 90 per cent of the data for 20 epochs and a batch size of 8.

2.4. Results

Once the training was complete, we noticed that while the model performed well on the train images (because of overfitting), the predictions on the test images were not good at all. Additionally, the validation was not working, and there was a warning that it is probably because the sample is small. After realizing that the generalization of the model was bad, we resort to image augmentation to create more samples from the 74 training samples provided.

2.5. Augmenting data and improving results

Having identified the issue of small data samples, we used a series of techniques to generate data from the samples provided using the *ImageDataGenerator* function in Tensorflow. For each image, we carried out six different augmentation techniques: rotation range, width shift range, height shift range, zoom range, vertical flip and horizontal flip. The ideas for these were inspired

by this tutorial [5]. This augmentation is done on the input images and masks at the same time. You can check the newly created images in this [folder](#) (specifically in “input_image” and “segmentation_seed”). You can also generate these images using the function below:

```
def generate_images():
    generators = [
        ImageDataGenerator(rotation_range=135),
        ImageDataGenerator(width_shift_range=.15),
        ImageDataGenerator(height_shift_range=.15),
        ImageDataGenerator(zoom_range=0.5),
        ImageDataGenerator(vertical_flip=True),
        ImageDataGenerator(horizontal_flip=True)
    ]

    prefix = ['rt', "ws", "hs", "zr", "vf", "hf"]

    for i in range(len(generators)):
        num_itr = math.ceil(74/batch_size)
        seed = 42
        img_gen = generators[i].flow_from_directory(
            'train2/input_image',
            class_mode=None,
            batch_size = batch_size,
            save_to_dir = "train2/image_gen",
            save_prefix= prefix[i],
            seed=seed)

        for img_batch in range (num_itr):
            img_gen.next()
        mask_gen = generators[i].flow_from_directory(
            'train2/segmentation_true',
            class_mode=None,
            batch_size = batch_size,
            save_to_dir = "train2/mask_gen",
            save_prefix= prefix[i],
            seed=seed)
        for mask_batch in range (num_itr):
            mask_gen.next()
```

Figure 4: Generating more training samples

Once this augmentation was done, there were 444 additional images for input image as well as the mask, bringing the total number of samples to 518.

With these new training samples, we trained the model again with the same hyperparameters, except we change the epochs to 40. After this training, the result improved considerably, yielding 95% on the validation set.

```
Epoch 34/40
52/52 [=====] - 112s 2s/step - loss: 0.0778 - accuracy: 0.9655 - val_loss: 0.1240 - val_accuracy: 0.9485
Epoch 35/40
52/52 [=====] - 108s 2s/step - loss: 0.0820 - accuracy: 0.9637 - val_loss: 0.1248 - val_accuracy: 0.9498
Epoch 36/40
52/52 [=====] - 102s 2s/step - loss: 0.0739 - accuracy: 0.9671 - val_loss: 0.1182 - val_accuracy: 0.9512
Epoch 37/40
52/52 [=====] - 112s 2s/step - loss: 0.0729 - accuracy: 0.9673 - val_loss: 0.1219 - val_accuracy: 0.9493
Epoch 38/40
52/52 [=====] - 115s 2s/step - loss: 0.0718 - accuracy: 0.9678 - val_loss: 0.1250 - val_accuracy: 0.9491
Epoch 39/40
52/52 [=====] - 109s 2s/step - loss: 0.0670 - accuracy: 0.9698 - val_loss: 0.1129 - val_accuracy: 0.9548
Epoch 40/40
52/52 [=====] - 100s 2s/step - loss: 0.0719 - accuracy: 0.9675 - val_loss: 0.1135 - val_accuracy: 0.9535
```

Figure 5: Training results

2.6. Prediction

After training for the last time, we saved the model so that we do not have to start training every time. For now, to train again, you call the train_model function in the code. Below are some of the predictions:

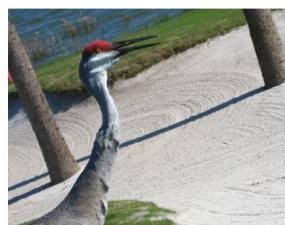


Figure 6: Sample prediction 1



Figure 7: Sample prediction 2

3. Idea for improvement

One idea we had was that could we do an ensemble of the results from GrabCut and U-Net to create an even better result? We did not explore this due to time constraints. Still, it would be interesting to see if doing a template matching over the two images and maintaining the pixels where the match is more substantial could improve the result across most of the images. Of course, some of them will fail like in the case of the figure below.



Figure 8: Sample prediction 2 (Middle: GrabCut result, Right: U-Net result)

When you do template matching in this case, the beak is going to disappear totally. But could the result improve overall? Or maybe template matching is not the correct way to do ensembling in this case. This requires further research.

4. Conclusion

Through this process, we learned a lot of lessons, foremost of which is that segmentation is not a trivial problem. The wide-ranging nuances in images make it hard to develop an algorithm that fits all situations. Secondly, as we worked on this assignment, it was clear that the idea of just copying and pasting code does not work most of the time. We learned to follow documentation and find ways of adapting code to suit our needs while acknowledging the effort others have put in. Finally, through this assignment, it becomes clear to us that computer vision has varying applications that can help us solve problems in different domains.

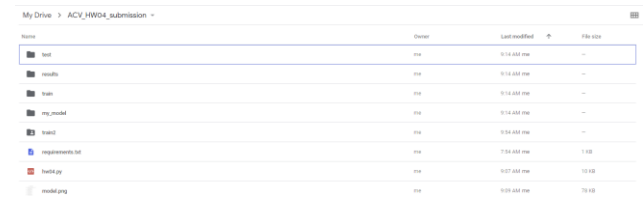
We are looking forward to making the best use of this knowledge.

References

- [1] G. Friedland, "Simple Interactive Object Extraction," *SIOX*. [Online]. Available: <http://www.siox.org/>. [Accessed: 13-Dec-2020].
- [2] A. Rosebrock, "OpenCV GrabCut: Foreground Segmentation and Extraction," *PyImageSearch*, 27-Jul-2020. [Online]. Available: <https://www.pyimagesearch.com/2020/07/27/opencv-grabcut-foreground-segmentation-and-extraction/>. [Accessed: 13-Dec-2020].
- [3] N. Tomar, "Unet Segmentation in TensorFlow," *Developer*, 07-May-2020. [Online]. Available: <https://idiotdeveloper.com/unet-segmentation-in-tensorflow/>. [Accessed: 13-Dec-2020].
- [4] "Image segmentation : TensorFlow Core," *TensorFlow*. [Online]. Available: <https://www.tensorflow.org/tutorials/images/segmentation>. [Accessed: 13-Dec-2020].
- [5] P. Patidar, "Image Data Augmentation- Image Processing In TensorFlow- Part 2," *Medium*, 20-Nov-2020. [Online]. Available: <https://medium.com/mlait/image-data-augmentation-image-processing-in-tensorflow-part-2-b77237256df0>. [Accessed: 13-Dec-2020].

Appendix

The folder structure in the code submitted was to follow the guidelines in the assignment (i.e src files should be in "src" and results in the "results" folder) and the augmented dataset was not included in the "src" files. To be able to run the code related to everything in this document, you can see this [folder](#) structure (noticed that the files were last updated before the deadline) with everything needed to run the following commands.



Name	Owner	Last modified	File size
test	me	0.00 MB	0.00 MB
results	me	0.00 MB	0.00 MB
src	me	0.00 MB	0.00 MB
src_model	me	0.00 MB	0.00 MB
test	me	0.00 MB	0.00 MB
requirements.txt	me	1.00 KB	1.00 KB
hw04.py	me	10.00 KB	10.00 KB
hw04.jpg	me	70.00 KB	70.00 KB

For one image, you can run the following command:

```
python hw04.py -i 2011_002851.jpg -of results
```

This assumes that you have the image and the result folder in the same file as the code.

For all images in a folder, you can run the following command:

```
python hw04.py -i input_images -of segmentation_generated -n True
```

This assumes that you have a "test" folder that contains input_image" and "segmentation_generated".