Samuel Bernsen

Assignment 1 Report: Water Body Segmentation

Introduction

Image segmentation is applicable to many fields including autonomous vehicles, robotics, surveillance, medical imaging, and remote sensing. It involves categorizing objects within an image and finding their associated boundaries. This project involves segmenting water bodies from the surrounding area using a random forest model and a custom U-Net architecture. The models extracted meaningful semantic information from the images and classified each pixel as water or background. Practically, environmental agencies could use a model like mine to identify raising or lowering water levels on specific bodies of water. Military personnel could also utilize binary masks to understand unknown enemy territory before embarking on a mission. I worked on this task because I have little experience with segmentation models and binary mask prediction seemed manageable as a first attempt. Also, the quality of the predicted masks is visually easy to evaluate. Initially, I attempted to implement a K-Nearest Neighbors classifier to segment the images; however, it did not finish training. Then I built a random forest model and a deep fully convolutional U-Net. Deep learning is not required for this task, but I prefer it to simpler classifiers because it is surprisingly much quicker to train and easier to implement. Overall, the maps generated from inference pass the eye test and achieve moderately high accuracy scores.

Data and Methods

The data used for training and validation is from a Kaggle dataset with a total of 2841 images and the same number of corresponding binary masks. A quick check through the folders revealed no obvious misclassifications by the dataset creator. Still, there were a few all-black images revealed when checking the n best and n worst-performing images. Every image was in JPEG format, and they varied dramatically in size.

I down sampled and standardized each image to 224x224 pixels to smooth the training process. Since there was more information available in most images, it would have been advantageous to down-sample less, but the accuracy was already high. Increasing the standard image size would take more computer resources for minimal benefit to the model's performance. The images were loaded as NumPy arrays and normalized by dividing by 255. For data augmentation, I used the ImageDataGenerator from Keras's preprocessing library because my previous augmentation method used for image classification did not work. Within the data generator, I included rotations, shifting, shearing, zooming, and flipping. This combination empirically worked best for my data.

The U-Net consists of multiple convolutional layers to perform down-sampling and transpose convolution to perform up-sampling in the spatial domain. My architecture is like the original U-Net developed by Ronneberger, Fischer, and Brox, but the number of filters per convolutional layer is different to decrease the computational load. Each convolutional layer uses a weight initializer which selects from the standard normal Gaussian distribution and uses same padding. I included the same residual connections proposed in the U-Net paper to preserve prior information and make backpropagation easier. The final layer assigns a class label to each pixel by choosing the class with the largest confidence. The sigmoid activation function on the final convolutional layer accomplishes this pixel-labeling

Results

The U-Net's performance is satisfactory with about 65% accuracy for the training and validation sets. I examined the 3 best, 3 worst, and two other randomly selected images to visually understand the model's performance on a subset of the data. To my eye, the model performed almost perfectly with no obvious calibration issues. Oddly, the random forest performed better with 82% accuracy but took over 7 minutes to train with only 150 images. The random forest classifier took longer to train than the deep U-Net and the K-Nearest neighbors classifier did not finish training after ten minutes so I decided not to

waste resources training it to completion. I believe the main culprits for longer training times are: 1. The need for feature engineering in basic ML models, 2. Lack of GPU acceleration, and 3. The lack of optimization techniques available for Sci-kit Learn-based implementations. Deep architectures generally perform well at universal function approximation because they can learn relevant features by themselves, whereas simple architectures may need feature engineering to achieve similar performance. Also, the Cuda programming library is highly optimized for Keras, TensorFlow, and Pytorch but not for Sci-kit Learn which helps speed up training. Lastly, built-in optimizers like Adam, RMSP, batch normalization, and stochastic gradient descent for TensorFlow (which drastically reduces the computational load) don't exist in Sci-kit Learn's toolbox. The compute times are less for the deep architecture, so the U-Net is preferable.

Figure 1: Random mask prediction

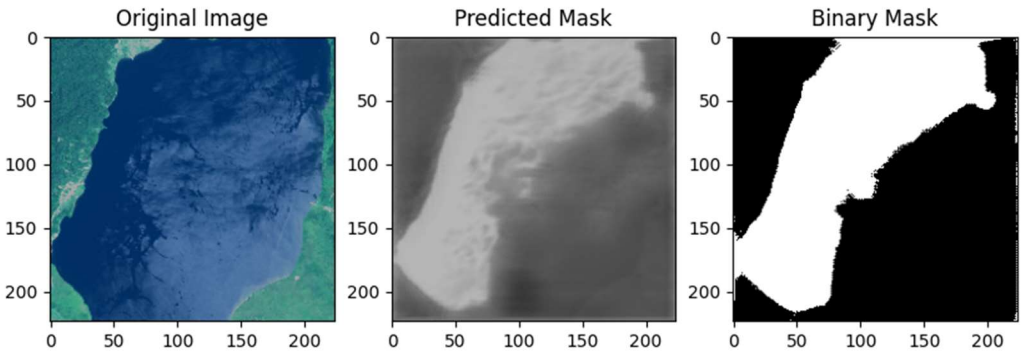


Figure 2: Accuracy During Training

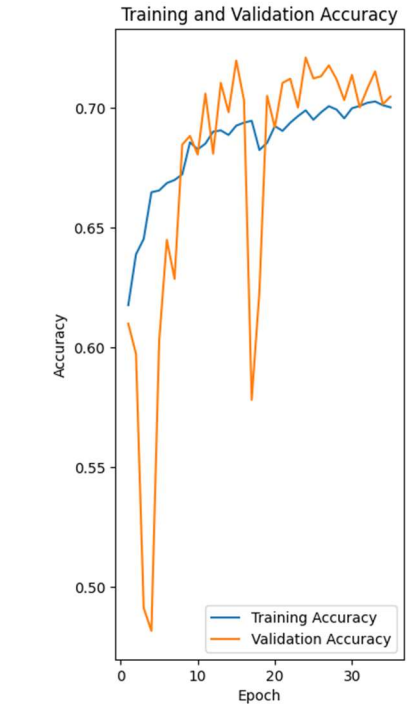


Figure 3: The Most Confusing Images for the U-Net

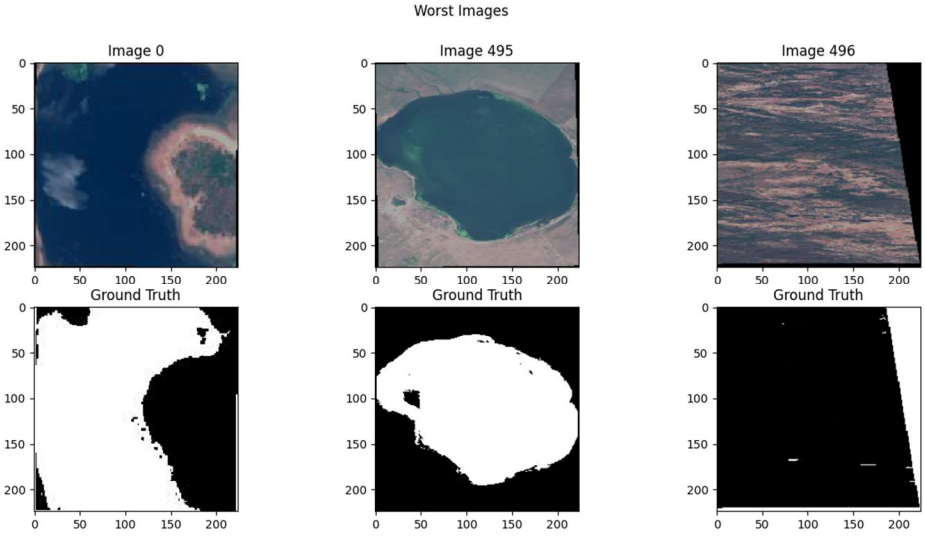
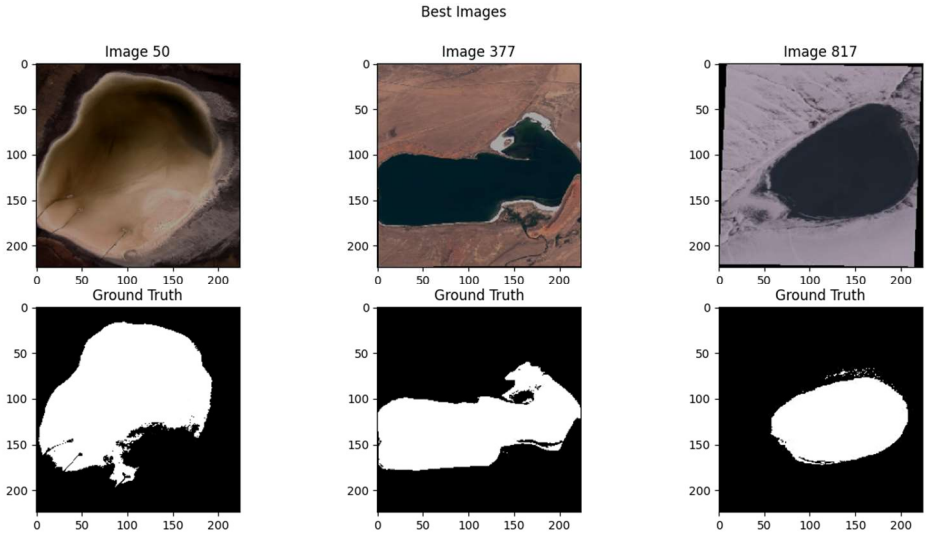


Figure 3: The Least Confusing Images for the U-Net



Discussion

Overall, my pixel-wise classification algorithms performed to my expectations. Visually inspecting the images and corresponding predicted masks gives me confidence in the U-Net architecture for water body segmentation. My architecture is based on the original version of the U-Net and thus did not need major tuning. I expect that increasing the number of possible classes would make this problem much more difficult and would require longer training. Future work would include a larger data set with more possible pixel classifications and more images in each class. I also want to experiment with a vision transformer to compare the results directly. Throughout my time experimenting with hyperparameters, I occasionally encountered odd behaviors. For this reason, I recommend this model be used in production with intermittent human error checking. My main takeaway from this assignment is that basic machine learning models can take longer to train in certain situations. Complex models do a great job of deriving accurate relationships between features without needing to perform tedious preprocessing on the data.