**Performing Semantic Segmentation on FloodNet**

Kyle Fletcher

Kaden Gryphon

**Introduction:**

This project uses the FloodNet Dataset. The FloodNet Dataset is about two-thousand aerial drone images taken after Hurricane Harvey. The goal of this dataset is to train a model that can help identify damaged areas for first responders to prioritize after a natural disaster.

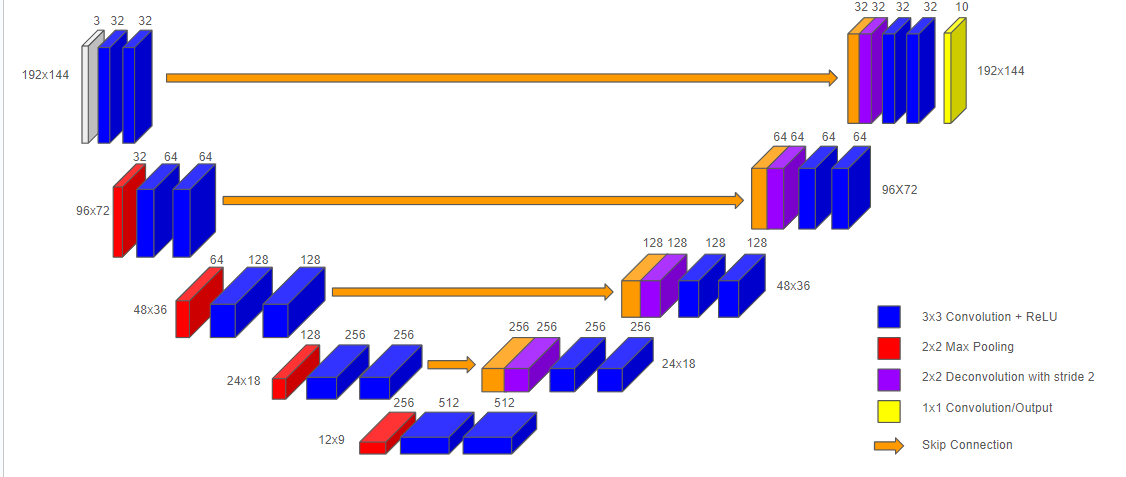
Each of the images are 4000x3000 pixels and are paired with a mask image that classifies each pixel into one of ten classes. The ten classes are: Background, Water, Building Flooded, Tree, Building Non-Flooded, Vehicle, Road Flooded, Pool, Road Non-Flooded, and Grass. Our goal was to create and train a model using a U-net like architecture to perform per pixel classification on the images.

**Architecture and Algorithms:**

Our dataset is already split into training, validation, and test data. The training data is composed of 1445 images, the validation data is composed of 450 images, and the test data is composed of 448 images. There are 2 main steps of preprocessing that need to be done. First the images are resized to 192x144. This is due to processing limitations given the devices we have. If the images are much bigger they begin to use up too much of the GPU. The other preprocessing step is normalizing the 0-255 rgb inputs to 0-1 floating point values for the original images to be passed through the network.

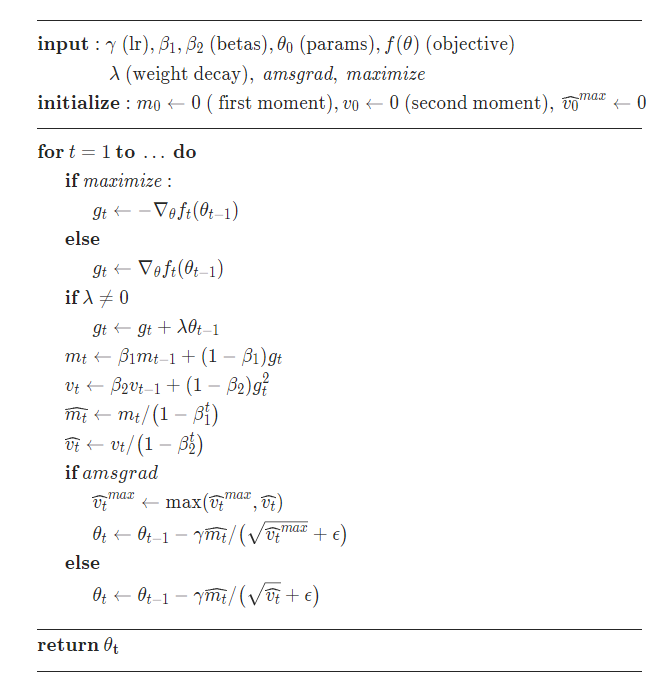
The neural network itself is implemented using PyTorch and is a fully convolutional network. It is made up of 3x3 convolutional layers, 2x2 max pooling layers, 2x2 deconvolutional layers with stride 2, skip connections, and a 1x1 convolutional layer at the end. Every convolutional layer is followed by a ReLU activation. The input, which is 192x144x3 goes through 2 convolutions of 32 filters each. Then it goes through a 2x2 max pool making the dimensions 96x72x32. Then it goes through 2 more convolutional layers of 64 layers each followed by another 2x2 max pool making the dimensions 48x36x64. Next it goes through another 2 convolutional layers with 128 filters each and another 2x2 max pool making the dimensions 24x18x128. After that it goes through 2 more convolutional layers with 256 filters each and a final 2x2 max pool making dimensions 12x9x256. This gets put through 2 more convolutional layers with 512 filters each making the data 12x9x512. After this is the first deconvolution layer with 256 filters bringing the dimensions to 24x18x256 which gets concatenated with the outputs of the 256 filter convolution layers in a skip connection. This is followed by 2 more convolutional layers of 256 filters. Then is another deconvolution with 128 filters, making the dimensions 48x36x128. This gets concatenated with the outputs of the 128 filter convolutional layers and is subsequently passed through 2 more convolution layers of 128 filters. This output is passed through another deconvolution of 64 filters making it 96x72x64, and is concatenated with the outputs of the 64 filter convolutions in another skip connection. 2 more 64 filter convolutions follow and the output is passed through a final deconvolution layer making the dimensions 192x144x32. This is concatenated with the outputs of the first convolutional layers and passed through 2 final 3x3 convolutional layers of 32 filters. Finally, the output is passed through a 1x1 convolutional layer of 10 filters to yield the output 192x144x10 corresponding to the 10 classes of the data.

Below is a full diagram of this architecture.



For the training process, the model is trained stochastically with mini-batches of size 64. It was observed that for a dataset so complex, overfitting easily becomes a problem after too many epochs. To combat this problem, the model uses a stopping condition for training. The model requires that the validation loss continue to improve each epoch. If the model fails to improve for 10 or more epochs, the model stops training and the version of the model with the lowest validation loss is what’s saved. We do set a max of 500 epochs but do not come anywhere close to that limit.

The model is trained each iteration using the ADAM optimizer. ADAM is a first order optimization method, meaning it considers the gradient of the loss of the network with respect to the trainable parameters in the network. ADAM utilizes 2 momentum terms so that update steps consider the previous direction steps to improve the stability of parameter updates. Adam is currently one of the most widely used and successful optimization algorithms for deep neural networks, which is why we chose it. We utilize the PyTorch implementation of ADAM. The ADAM algorithm is given below.

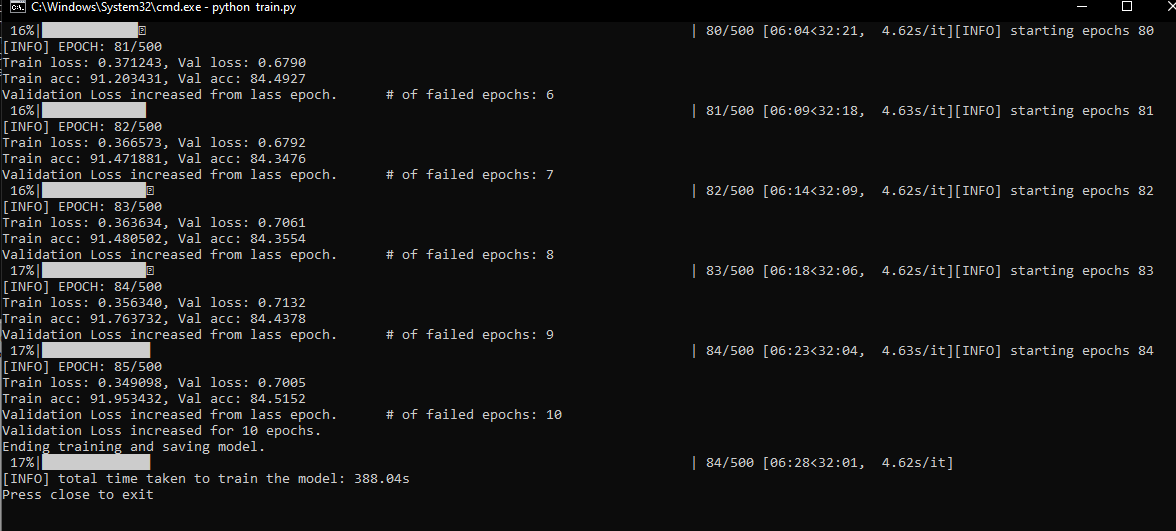


In our implementation we use a learning rate of 0.001, a first momentum parameter of 0.9 and a second momentum parameter of 0.999, which are common choices for these parameters. We don’t use any weight decay or use the amsgrad method.

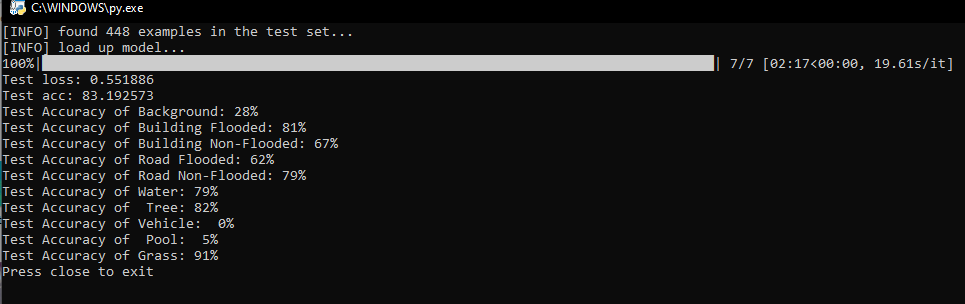
**Results:**

Overall the results were pretty good considering the smaller dataset size complexity of the images, and the challenge of running the training on a home computer. The training ran for 84 Epochs before ending early due to failing to improve the validation score ten times in a row.

The training was done on a Ryzen 7 3700X 8-Core CPU and an RTX 3080 GPU. The longest part of the process was running the transform functions on the original images, this took about 10 minutes and is why we started saving a copy of the transformed images. Training the model took about 7 minutes to run using about 60%CPU 10%GPU and 5GB or RAM. Testing took 2 minutes, but used far more resources, 5%CPU, 50-100%GPU, and 22GB of RAM.

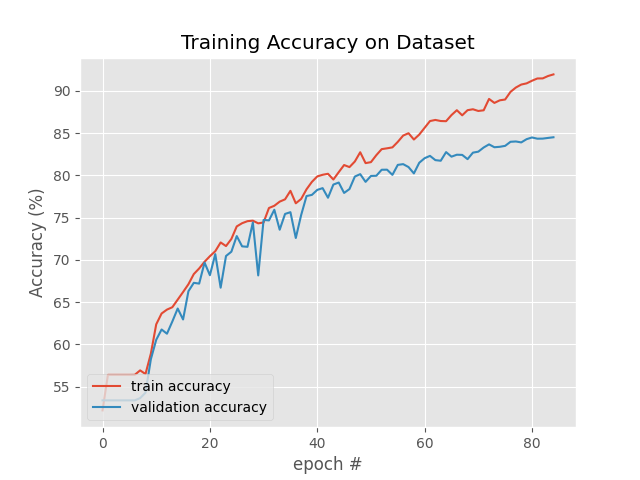


-Screen shot of the training output



-Screen shot of the testing output

We recorded the loss and accuracy of the model after each epoch on the training and validation datasets. While the validation leveled off the training seemed on track to go farther. This shows that the model could risk being overtrained if further epochs were run.

****

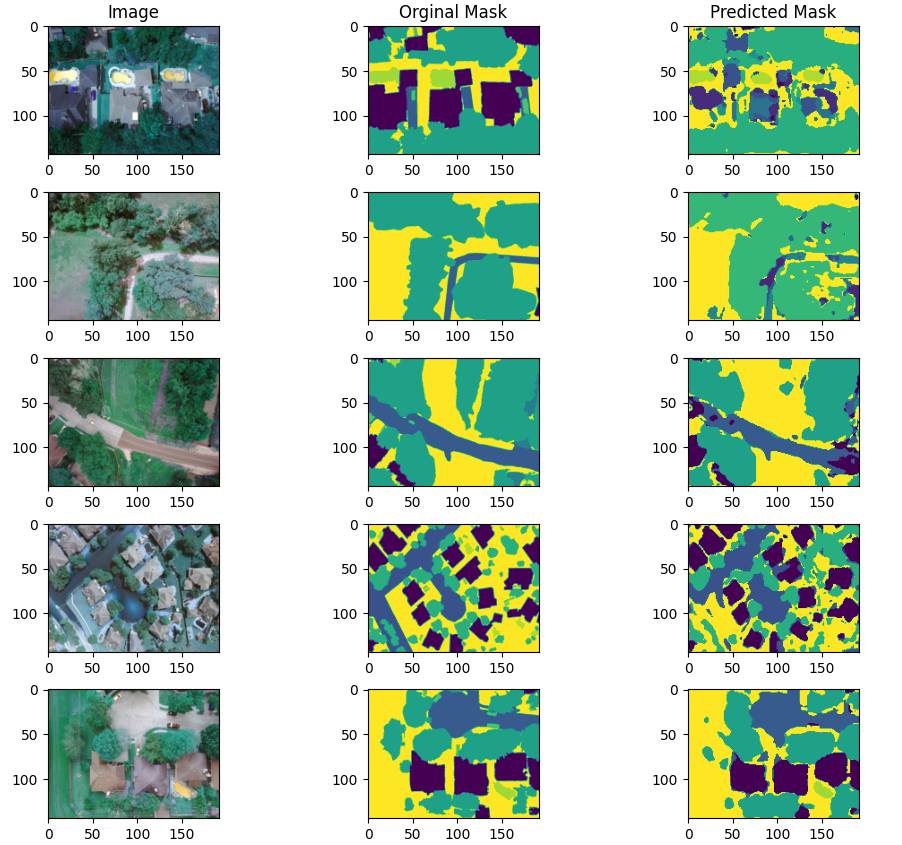
-Graphs for the loss and accuracy for each epoch

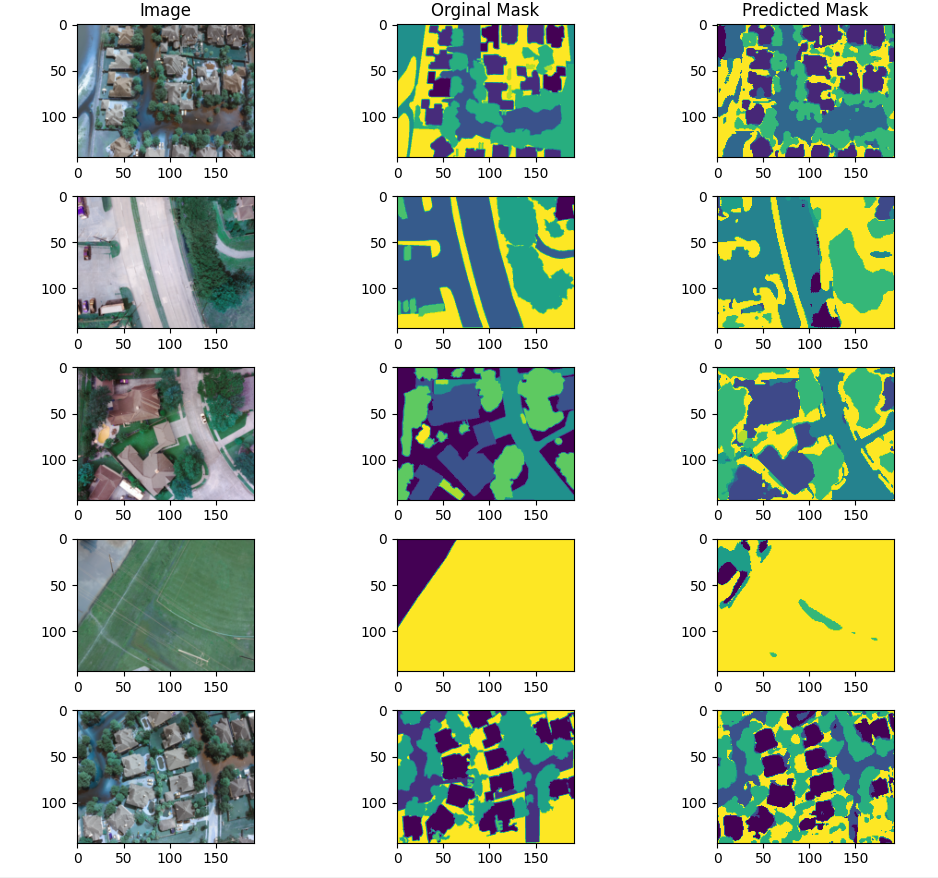
| Total Loss | 0.551 |
| --- | --- |
| Total Accuracy | 83.19% |
| Accuracy of Background | 28% |
| Accuracy of Building Flooded | 81% |
| Accuracy of Building Non-Flooded | 67% |
| Accuracy of Road Flooded | 62% |
| Accuracy of Road Non-Flooded | 79% |
| Accuracy of Water | 79% |
| Accuracy of Tree | 82% |
| Accuracy of Vehicle | 0% |
| Accuracy of Pool | 5% |
| Accuracy of Grass | 91% |

-Table for the total loss and accuracy and the accuracy for each class

After tweaking the model and trying a few different designs we achieved an accuracy of 83.19% on the test images. For many of the classes the model had fairly good results like the flooded building and roads. Some classes had quite a few issues. Classes like vehicles and pools did the worst by far. For vehicles we believe that the lower number samples as vehicles were not as common in the training images and that with the lower resolutions the vehicles were usually very small.

The other class that struggled was “Background” I believe that this is due to background often being things like trees and grass and it gets classified as those instead.





-Some output plots comparing the original images with the original maks and the predicted masks

Looking at our predicted masks, it can be seen that they are often close to the original masks. It does show something the model struggles with. While it often finds objects like houses and roads it often struggles to get a clear outline and the edges of objects tend to be a mix between the two classes. The model also seems to do worse with large areas of the same class. You can also see an example of where the model does not see the cars.

Overall while the model misses small objects like cars and struggles to get clean outlines of the objects it does a fairly good job identifying objects and their rough locations in the images.