Tony Albanese
GTECH-731
Final Project
05/20/2021

Improving Neural Network Training for Floodwater Segmentation using the Sen1Floods11 Dataset

Problem

The goal of this project was to improve floodwater recognition in a fully convolutional neural network (FCNN) trained on the Sen1Floods11 dataset. This dataset was published in 2020 in order to "… assist efforts to operationalize deep learning algorithms for flood mapping." [1]. It consists of three main categories of water imagery: permanent water, hand labelled flood images, and weakly supervised algorithmically labelled flood images. The flood imagery comes from 11 different geographically diverse flood events – hence the 11 in Sen1Floods11. Due to the initial poor results from the original researchers attempting to train a neural network on the permanent water dataset and the time limited nature of this project, I did not attempt any work with the permanent water imagery. The hand labelled data consists of a label set - 446 512x512 GeoTIFF files labelled water/no water/no data – and then coincident imagery from Sentinel-1 and Sentinel-2. The machine labelled data consists of 4,385 512x512 GeoTiff files of Sentinel-1 imagery and then two coincident sets of labels – one generated using an Otsu thresholding algorithm on the VH band on Sentinel-1 imagery and one generated using a combination of Normalized Difference Vegetation Index and Modified Normalized Difference Water index on Sentinel-2 imagery. All Sentinel-1 imagery has been subset to include only VV and VH bands.

Along with the publication of the original dataset, the authors of the dataset also included results from preliminary training on an unoptimized FCNN which showed that, when trained on the weakly supervised dataset with Sentinel-1 imagery as the input and Sentinel-2 derived water labels as the output, the FCNN was able to achieve results on floodwater segmentation, as assessed by mean intersection-over-union (IOU) analysis on hand labelled data, that were equal to or greater than Otsu thresholding. Based on the early success of this FCNN, this project aimed to improve training results against both Otsu thresholding and hand-labelling on the dataset through optimization of the FCNN training procedures and architecture.

Conceptual Model

The basis of this model is the fully convolutional neural network architecture. I will break this down into its component parts, though a thorough explanation of the workings of an FCNN is beyond the scope of this paper. For further explanation, I highly recommend the books *Grokking Deep Learning* by Andrew Trask [2] and *Deep Learning with PyTorch* by Stevens, Antiga, and Viehmann [3]. These were both invaluable resources while working on this project.

A neural network is a mathematical construct which takes a set of inputs and a set of outputs and attempts to learn to transform the inputs to the outputs. It does this using a series of layered calculations, wherein features from the input are transformed by the multiplication of a weight and

addition of a bias and then put through a non-linear activation function that passes the output to the next layer, wherein the multiplication, addition, and activation process repeats. This movement of data through the layers eventually returns an output. The difference between the expected output and the output from the neural network is calculated, and that difference is propagated backwards through the network to modify the weights and biases. As inputs repeatedly move through the network and are compared to outputs, the back propagation process works to change the weights and biases in order to transform the input more closely to the expected output. As an oversimplification, you can think of a neural network as a large regression machine attempting to find the relationship between each part of the input and each part of the output [2].

A convolutional neural network uses convolutions to extract features from an input image and pass them through the neural network, often changing the size and dimensionality of the input. For example, a strided convolutional layer could be used to down sample an image – making it smaller – combined with a max pooling function – giving us only the highest values – which would produce a layer giving us a compressed image that may have desired features highlighted. Convolution can also be used to increase the number of channels in an image, allowing different aspects to be highlighted for "consideration" by the activation functions of the neural network. While convolutional neural networks usually down sample their input into a smaller size or dimension, for example taking an image of a pig and outputting the one-word label "pig", a fully convolutional neural network gives an output equal to the original input size [3].

In terms of this project, I utilized three different FCNN architectures in order to attempt to train a network which was able to identify water pixels. The inputs were all Sentinel-1 imagery, and the outputs were images with each water pixel assigned a value of 1 and each non-water pixel assigned a value of 0. Though I utilized different FCNNs, all of them were relatively "deep" neural networks between 18 and 50 layers which utilized skip connections in order to back-propagate the gradient through shallow layers. A skip connection, originally described with the publication of Resnet [4], passes the results of a layer forward in a network beyond the layer immediately in front of it. This works to both decrease information loss due to pooling layers as well as allow for the weight modification gradient calculated from the output loss to propagate to the beginning of a deep network, since the "skips" work in both directions.

I utilized FCNNs for two reasons. Primarily, it was because I am still a deep learning novice, and it is the architecture the researchers who created the Sen1Floods11 dataset utilized. Since my goal was to build upon their work, it made sense to utilize the same architecture as them. Additionally, since I viewed this as in image segmentation problem and FCNNs have shown excellent results for image segmentation for a diversity of datasets, from regular object recognition to tumor identification [5] to cloud segmentation from remotely sensed imagery [6], it made sense to focus on FCNNs.

Methodology

Implementing FCNN training in PyTorch

I began this project by implementing a neural network training script using PyTorch, one of two leading Python libraries along with TensorFlow for facilitating deep learning. I chose PyTorch because it

was utilized by the researchers who published the Sen1Floods11 data and frankly, it is quite a bit easier to install on Windows than TensorFlow. A neural network training script using PyTorch consists of implementing a dataset class which identifies all the files to be used for training and any transformations that should be applied to these files, a data loader which loads the data on demand for training, a training loop which trains the neural network, and a validation loop which tests the neural network against data it has not been trained on. There is a separate test loop for final results testing as well.

In order to implement the dataset class, I had to first split the data into training, validation, and test groups. The training group provides the input/output for the neural network to actively train on, while the validation group is used to test the neural network after each training loop and determine if the network is overfitting the training data. Overfitting means that the network is not learning to solve the general problem of input -> output, but instead learning to solve the input -> output problem only for that specific training set. The test group is kept entirely separate from the training process and is used to test the results of the network after the training is complete. I split the data along standard proportions of 0.6 training, 0.2 validation, and 0.2 testing. Additionally, I made sure to leave data from Bolivia out entirely, as the original researchers maintained this as an entirely separate testing set and I aimed to compare my results to theirs. When splitting the data, I measured the number of water pixels in each image based on the provided labels and ensured that the distribution of water pixels in each split matched the distribution of water pixels in the whole group.

Once I had the splits saved as csv files, I was able to implement a dataset class based off the dataset class provided by the original researchers. This class can be found in flooddata.py. While I maintained the random cropping and flipping transforms of images when loading the data that the original researchers implemented – these transforms augment the dataset by creating random variations on the images and allow the network to train on smaller, less memory intensive images – I rewrote the rest of the dataset class so that it could be used in parallel by a PyTorch data loader. This allowed for significant processing speed increases, as you can set a number of “workers” in PyTorch to run image batches concurrently.

With the parallel friendly dataset implemented, I wrote the training and validation loops and nested them in an epoch loop so that they could run alternately. I also added simple logging using the TensorBoard library so I could track network output as it trained. I put the test loop in an entirely different file named test_model.py. Unlike the Sen1Floods11 researchers who included many global variables in their implementation, including a tragic abuse of “i” as a global variable, I functionalized each part of the training script so that there would be no namespace conflicts.

Hyperparameter Search

Now that I had the neural network training script fast and functional, I began a hyperparameter search. While the individual weights and biases that make up a neural network are parameters, the numbers that you feed into the various training components are the hyperparameters. Properly optimizing hyperparameters can make an enormous difference in the ability of a neural network to converge on the correct parameters for your model. Starting with the same binary cross entropy (BCE) loss function, Adamw optimizer, FCN ResNet50 network, and cosine annealing scheduler with warm restarts as the original researchers, I performed a systematic search on the following hyperparameters:

Batch Size – 4, 8, 16, 32 – how many images to process before updating model parameters
Learning Rate – .1 to .00001 – how much to update model parameters
Beta 1 and Beta 2 from 0 to 1 – gradient average coefficients used to calculate gradient momentum
Weight Decay from 0 to 1 – keeps the weights small and prevents overfitting
Optimization Weights from 1 to 10 – how much to penalize wrongly identifying a class

I performed this search using the Ray library, a multiplatform software library designed for hyperparameter optimization. The Ray library allowed me to write a function which called my neural network training function with different variations on all of the aforementioned hyperparameters. It would run each network variation for up to 15 epochs (15 training + validation loops), checking results against previous runs to see if results were improving or declining. In the case of decline, the run would be halted and a run with different hyperparameters would begin. If a model increased performance, a copy of the model parameters would be saved. Since early tests showed much greater success with the weakly labelled data over the hand labelled data, this search focused largely on the former.

Using the Ray tuner I trained hundreds of models using variations on these hyperparameters, searching for the greatest loss minimization in 15 epochs. From these tests I found an optimal batch size of 16, a learning rate of .0005, beta 1 of 0.2 and beta 2 of 0.66, weight decay of 0.003, and an optimization weight of 1. Unfortunately, Ray support on Windows is experimental, and while it saved its search findings to hundreds of csv files, the TensorBoard logging feature failed and we do not have any nice graphs to show this search.

Loss Function Search

Following the hyperparameter search, my results had improved but not significantly. After training on the S1 weakly labelled dataset prior to the hyperparameter search the model achieved mean IOU of 0.19 on the validation set, following the hyperparameter search I was able to increase mean IoU to 0.21. Guessing that the way loss was being calculated may not be optimal for this task, I then proceeded to test different loss functions that are commonly used for semantic segmentation tasks, Dice Loss and Focal Loss [7]. Focal loss has been found to work for imbalanced data sets, which we have – there are far more non-water pixels than water pixels and the distribution is heavily skewed towards images with relatively few water pixels. Dice loss is a metric designed specifically to test segmentation results. I also tested IoU loss since the end goal of our problem is to maximize IoU. The implementations of these loss functions were adapted to work with Sen1Floods11 data from [8] and can be found in losses.py

After performing another hyperparameter search with these loss functions, first finding the most effective learning rate for each loss function and then testing each one over 50 epochs, I found that changing the loss function did not significantly improve training or validation results. While Focal, Dice, and BCE all converged to similar mean IoUs of 0.21, the IoU loss function was radically unstable and tended to collapse dramatically at 40 epochs. Since none of these loss functions offered improvements on BCE, and BCE is faster due to how it is implemented in PyTorch, I stuck with binary cross entropy as a loss function.

FCNN Model Architecture Search

After failing to find a more useful loss function, I moved on to testing model architectures. I had begun my optimization quest using a fully convolutional ResNet50 architecture that comes as part of the PyTorch torchvision library. This was the network used by the Sen1Floods11 researchers to test their data, and since it is designed to provide high quality results on image segmentation tasks [9], I had stuck with it. After hitting a wall in increased training performance, I decided to try two other architectures. The first was a variation on ResNet50 known as DeepLabV3 ResNet50 which was designed to improve semantic segmentation results over ResNet50 [10]. DeepLabV3 ResNet50 has been shown to increase mean IoU by 6% in semantic segmentation tests over Resnet50 [9]. DeepLabV3 ResNet50 is also conveniently pre-built as a PyTorch torchvision model and can be loaded with a single line. In testing DeepLabV3 I was able to achieve an increase of mean IoU on the S1 weak validation set up to 0.24. However, DeepLabV3 comes with serious performance drawbacks. It is very memory intensive and takes significantly longer to run a training and validation epoch than ResNet50 on my limited hardware. Saving the results of the model uses 150 mb – that's 150mb of just saved weight and bias parameters. While the results were promising, due to time and hardware constraints I ceased testing DeepLabV3 ResNet50.

For my final attempt at training optimization, I switched to using an 18-layer U-Net architecture for image segmentation implemented from [6]. A U-Net is similar to ResNet in that they both use skip connections to propagate data from shallow to deep layers, however a U-Net uses a symmetrical down sampling – up sampling scheme. Each down sample convolution is passed down to the next down sampling layer but also passed across to its up sampling pair. U-Nets have shown success in semantic segmentation [6]. While the early epoch training results were not as good as either ResNet50 variant, this model offered significant performance increases due to fewer layers and lower memory consumption. This allowed to me to run tests going into hundreds of epochs. Additionally, the lower memory requirement for the saved model parameters (2mb vs 150mb) allows me to easily operationalize the model and send it to others for use. By running the U-Net model for 80 epochs, I was able to achieve equivalent results on the S1 weak dataset compared to DeepLabV3 ResNet 50. I obtained similar equivalent results by running the U-Net model on the S2 weak dataset for 160 epochs.

Testing

In order to test the models I had generated, I ran each model with the complete set of S1 Hand labelled data, testing for mean IoU and mean per-pixel accuracy. Additionally, I generated sample images to show the difference between the neural network labelled data, the hand labelled data, and the algorithmically labelled data. See the appendix for these images. In testing, I discovered that due to a typo in the earlier version of my neural network script I had failed to properly save my DeepNetV3 and ResNet 50 models, and those models were unusable. While I learned a harsh lesson about checking results along the way, I believe that based on IoU recorded while training the models, the U-net results I produced are equivalent to the best DeepNetV3 I had produced. I also found that, like many things in the example code provided by the original researchers, the commission and omission calculations provided did not work properly. As I discovered this too late to fix, commission and omission error rates will be omitted form the results. Finally, in testing I also discovered that when calculating mean IoU in my training script, I had been dividing it twice, leading to smaller mean IoU results than what should have been. While again it is too late to go back and recalculate mean IoU for all the training epochs, the

results are still valid as numbers showing relative improvement and relation to each other. They should be around 4x higher depending on batch size and size of the dataset being tested. I was able to fix this error for my testing loop, giving accurate mean IoU results.

Implementation

My project is implemented as a set of python scripts. All scripts were run using Python 3.7 for Windows. All libraries were up to date as of May 1, 2021. The Ray library used was a nightly build as of May 10, 2021. The scripts:

neural_network.py – The main training script for the neural network, including model instantiation and the epoch loop. This is also where I utilized Ray tune for hyperparameter testing.

losses.py – A library of three loss functions implemented as Torch.nn classes that I tested with my model. Adapted to work with Sen1Floods11 data from [8].

tests.py – A library of functions for assessing image segmentation performance published by the Sen1Floods11 researchers adapted for use with my training script.

nets.py – Contains the U-net class definition as well as a helper function for converting batch norm layers to group norm layers. U-net adapted from [6], helper function from [1].

get_distributions.py – A script to create training, validation, and testing splits with equivalent water pixel distributions between each split.

flooddata.py – PyTorch dataset class set up to allow for parallel execution.

All scripts assume that they are in the same directory and that the Sen1Floods11 data has been loaded from the google cloud storage address `gs://senfloods11/` to a subfolder called “files”. Note that some file paths have been hardcoded due to issues with parallel operations running within Python on Windows, otherwise relative paths are preferred.

Results

Using the much smaller U-Net architecture than the ResNet50 FCNN utilized by Bonafilia et al, I was able to get equivalent or improved results segmenting flood water imagery, though these results come with a large caveat. Bonafilia et al were able to split their imagery into permanent, flood, and all water categories for assessment. These splits do not exist in the publicly available dataset. While there is permanent water in the dataset, it does not seem to geographically overlap with the data labelled “flood_events”. I reached out to the researchers for clarification on this along with other questions about the data in early May and never received a response. Going forward, I am assuming that the images provided in “flood_events” represent the “floodwater” dataset, though they may represent the “all water” dataset. Additionally, as mentioned in methodology section, I was unable to calculate commission and omission rates to compare against the original research due to time constraints. My

mean IoU and per-pixel accuracy results for predicted labels on the Sentinel 1 Hand Labelled dataset using a U-Net based FCNN were:

Hand Labelled Test	Test IoU	Test Acc
S2 Weak (160 epochs)	0.332	0.9576
S1 Weak (80 epochs)	0.2902	0.9321
S1 Hand (110 epochs)	0.3072	0.9549

The original research results using a ResNet50 based FCNN:

Hand Labelled Test	Test IoU
S2 Weak (200 epochs)	0.3389
S1 Weak (3 epochs)	0.2422
S1 Hand (100 epochs)	0.2421

I was able to achieve near equivalent results using a much more compact neural network when trained on Sentinel 2 derived labels, and better results when using Sentinel 1 derived hand or machine labels.

When testing on the 15 sample Bolivia dataset, which was entirely left out of any training, validation, or testing splits, I observed significant improvements. It should be noted that this is quite a small sample size.

Hand Labelled Bolivia	Test IoU	Test Acc
S2 Weak (160 epochs)	0.4431	0.9121
S1 Weak (80 epochs)	0.6560	0.9405
S1 Hand (110 epochs)	0.4083	0.9105

The original research results:

Hand Labelled Bolivia	Test IoU
S2 Weak (200 epochs)	0.2738
S1 Weak (3 epochs)	0.3296
S1 Hand (100 epochs)	0.2905

Example images generated by the S2 Weak, S1 Weak, and S1 Hand trained networks are included in the appendix.

Discussion and Conclusion

This project was mildly successful. By performing an exhaustive hyperparameter search as well as partial searches for other loss functions and FCNN architectures, I was able to show that FCNN training on the Sen1Floods11 data set could be improved using a much smaller FCNN than the original FCNN researchers utilized. The results from the Bolivia dataset were promising, as those showed dramatic improvements in water segmentation IoU. These results should be viewed with the knowledge that the sample from Bolivia is only 15 image/label pairs. To me, the best improvement I made was

showing that a much smaller architecture could be used to get near equivalent or better results. Training time was decreased by an order of magnitude by switching to an 18-layer U-net over a 50-layer ResNet. By further optimizing the U-Net, I believe I could increase mean IoU on hand labelled data further.

While I think that there are further optimizations to be made that could increase FCNN training performance on this dataset, I am not sure that an FCNN for semantic segmentation – even the more modern and lighter U-Net implementation - is the best approach to the problem of automatically labelling floodwater data. I am still a deep learning novice and what follows is pure postulation, but I believe that water segmentation, particularly floodwater segmentation, is a different enough problem from what FCNNs for semantic segmentation have been optimized to solve that they are not the ideal solution. Typically, FCNNs for semantic segmentation are used for identifying whole objects – bicycles, tumors, cars, pedestrians, etc. These are all objects where bounding boxes can be established and outlines can be drawn. FCNNs are great at this task. Floodwater in a satellite image is a not a whole object, but an expression of the relationship between ground water, surface water, geomorphology, and the built environment. There is no one shape of water – it comes in channels, pools, rivulets, and any number of other shapes. Pools can be nested within other pools with a little island right in the middle. There is not always an outline edge to find or convenient bounding box to draw, so many of the high order features extracted by an FCNN such as edges and shapes are not conducive to determining probability of a pixel or group of pixels being water. In this study, I found that the images created by the neural networks captured the large, easily identifiable shapes very well, but the neural network labelling fell apart when it had to deal with scattered ponds, or thin channels, or really anything nested. Channels that that to a human eye were obviously connected would be frequently cut off by the network labelling, since it contains no concept of water as a thing that flows.

Moving forward from FCNNs for image segmentation of floodwater, there are two architectures that I would like to investigate as I believe their goals are more in line with the task at hand. The first is a Conditional Generative Adversarial Network (cGAN). This is the same type of network used in creating deepfakes as well as the pix2pix project [11], otherwise known as the project that can take a drawing of a handbag and turn it into a picture of a handbag, but also that handbag is made of cats [12]. The goal of a cGan is not to identify/segment every object in an input, but instead to learn as direct a translation as possible from input to output. If we look at the problem of floodwater labelling not as semantic segmentation, but instead a problem where we have a satellite image, and we want it to be that image but all the water, this translation approach makes intuitive sense. One benefit of a cGAN over an FCNN is that in an FCNN, the loss function is static, while in a cGAN it is dynamically produced by the relationship between the generator and discriminator networks. Unfortunately, I did not have time to test this approach.

A second approach I would like to investigate is using a Capsule Network for floodwater segmentation. A Capsule Network is like a Convolutional Neural Network with two key differences – it maintains spatial relationships between pixels and dynamically routes information through the network as opposed to pushing everything forward and having it be filtered by non-linear activation functions [13]. I am interested in this architecture primarily due to its ability to maintain spatial relationships, as recognizing that a channel in one part of an image could be connected to a channel in another part of an

image could increase the ability of the network to output accurate water labels. Again, I did not have time to test this approach.

After improving labelling ability, the next step would be to include a temporal component, perhaps combined with meteorological data, allowing for temporal predictions. If such a system could be successfully implemented then highly detailed, localized, weather dependent flood prediction maps could be produced.

To summarize, I was able to incrementally improve training performance on an FCNN semantic segmentation model for identifying floodwater while making the output model smaller and faster. While this training could be further improved, I believe that examining architectures other than FCNNs would yield the best results towards the goal of being equivalent to hand-labelled data.

Works Cited

[1] D. Bonafilia, B. Tellman, T. Anderson and E. Issenberg, "Sen1Floods11: a georeferenced dataset to train and test deep learning flood algorithms for Sentinel-1," *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW),* pp. 835-845, 2020.

[2] A. Trask, Grokking Deep Learning, USA: Manning Publications Co., 2019.

[3] L. A. a. T. V. Eli Stevens, Deep Learning with PyTorch, USA: Manning Publications Co., 2020.

[4] X. Z. S. R. J. S. Kaiming He, "Deep Residual Learning for Image Recognition," *arXiv:1512.03385,* 2015.

[5] Y. B. F. P. A. P. N. K. D. T. Shervin Minaee, "Image Segmentation Using Deep Learning: A Survey," *arXiv:2001.05566,* 2020.

[6] M. Cordeiro, "Creating a Very Simple U-Net Model with PyTorch for Semantic Segmentation of Satellite Images," Medium.com, 5 April 2020. [Online]. Available: https://medium.com/analytics-vidhya/creating-a-very-simple-u-net-model-with-pytorch-for-semantic-segmentation-of-satellite-images-223aa216e705. [Accessed 10 May 2021].

[7] S. Jadon, "A survey of loss functions for semantic segmentation," in *2020 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)*, 2020.

[8] "Loss Function Reference for Keras and PyTorch," [Online]. Available: https://www.kaggle.com/bigironsphere/loss-function-library-keras-pytorch. [Accessed 15 May 2021].

[9] "Torchvsion.Models," PyTorch, [Online]. Available: https://pytorch.org/vision/stable/models.html#semantic-segmentation. [Accessed 10 May 2021].

[10] L.-C. Chen, George Papandreou, Florian Schroff and H. Adam, "Rethinking Atrous Convolution for Semantic Image Segmentation," *CoRR,* vol. abs/1706.05587, 2017.

[11] P. a. Z. J.-Y. a. Z. T. a. E. A. A. Isola, "Image-to-Image Translation with Conditional Adversarial Networks," *CVPR,* 2017.

[12] C. Hesse, "Image to Image Demo," 19 Feb 2017. [Online]. Available: https://affinelayer.com/pixsrv/. [Accessed 15 May 2021].

[13] N. F. G. E. H. Sara Sabour, "Dynamic Routing Between Capsules," *arXiv:1710.09829,* 2017.

Appendix

S1 Weak Trained Model

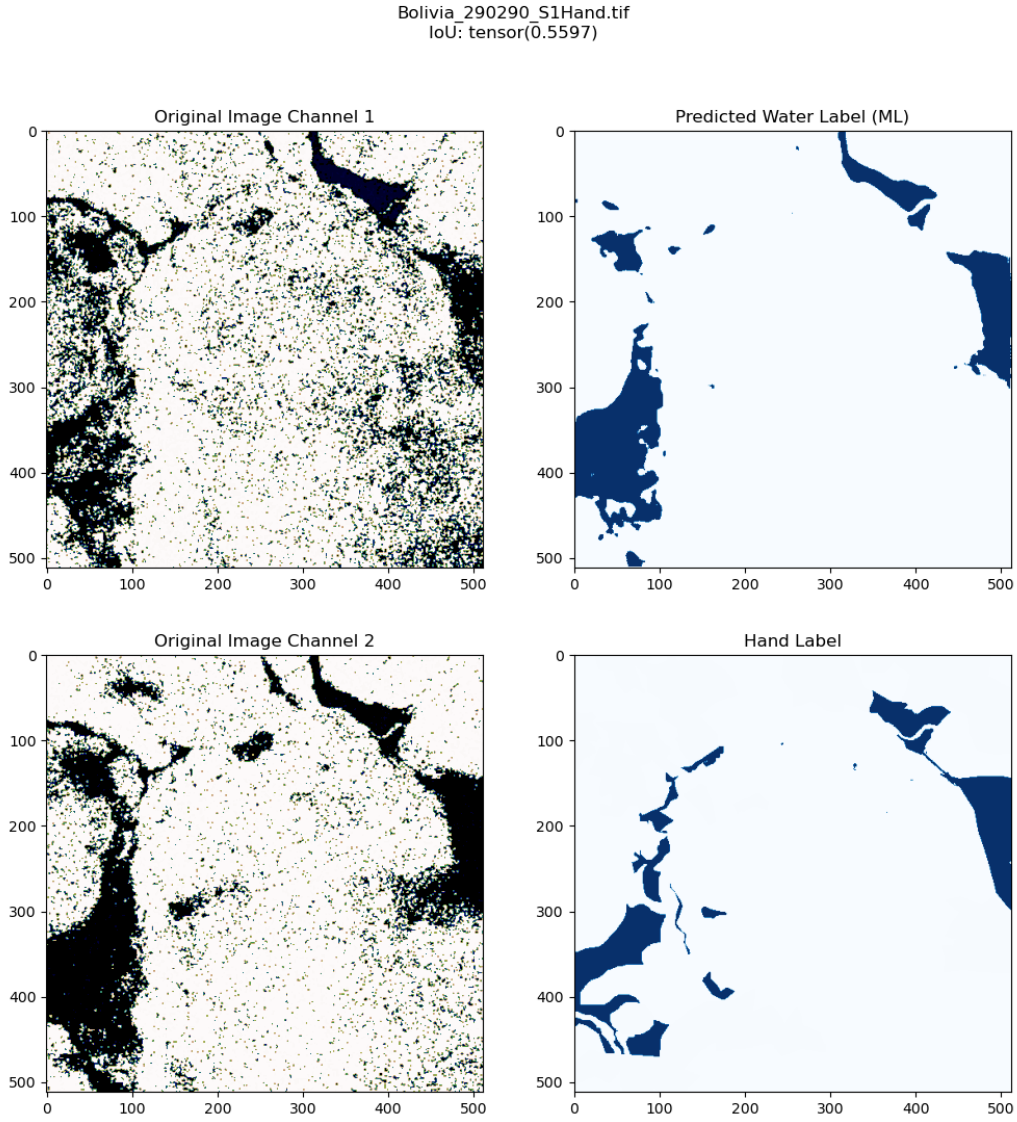

Figure 1*. This shows the original S1 input imagery on the left, with the two sets of water labels on the right. Upper right is the output from the neural network trained on S1 Weak data, lower right is the provided hand label. Note the loss of detail in the neural network produced labels. Imagery is from the Bolivia data subset.*

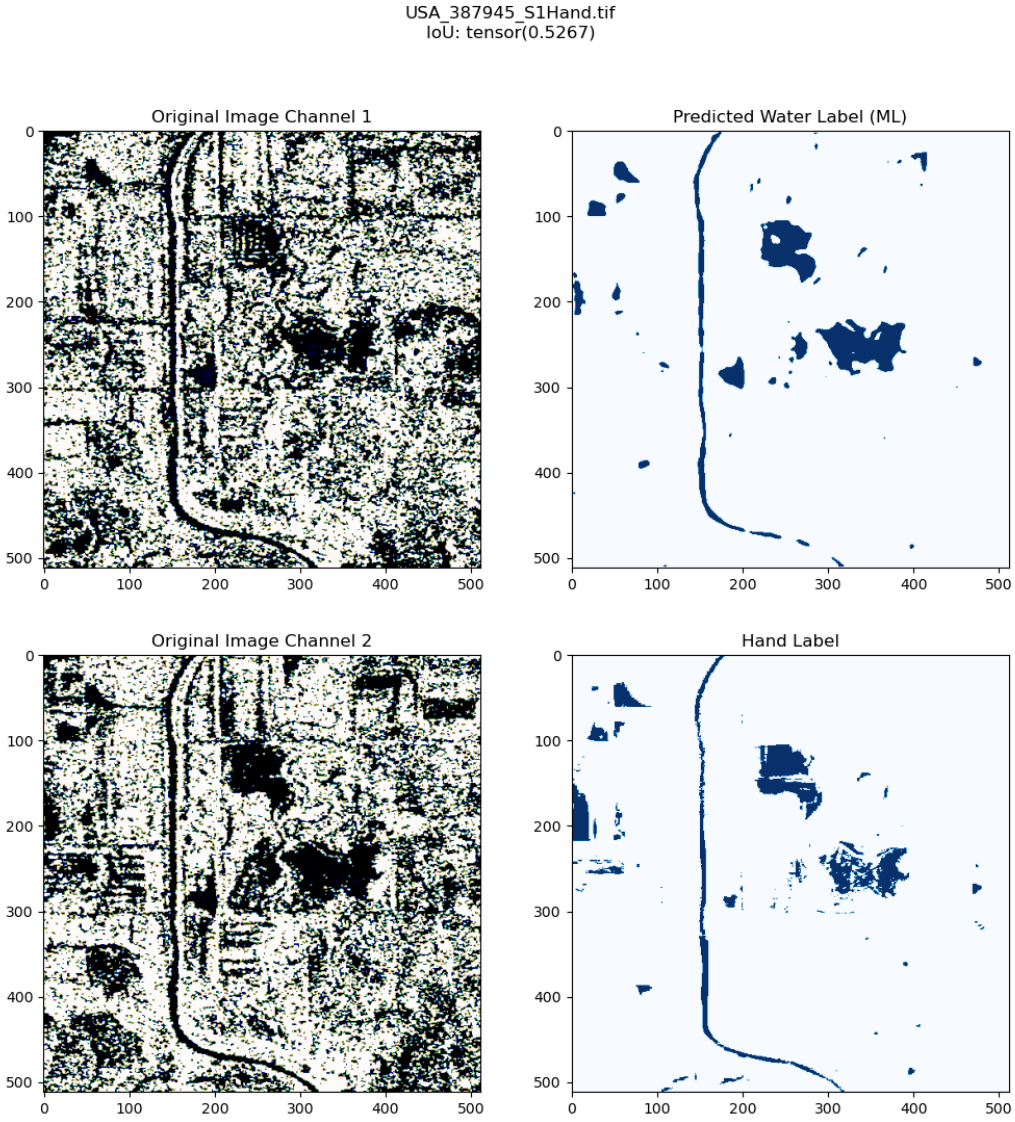

***Figure 2**. Prediction from the S1 Hand Labelled data subset. Again, note the loss of detail in the S1 Weak trained neural network produced imagery. See the failure to recognize a continuous channel in the bottom center of the predicted label.*

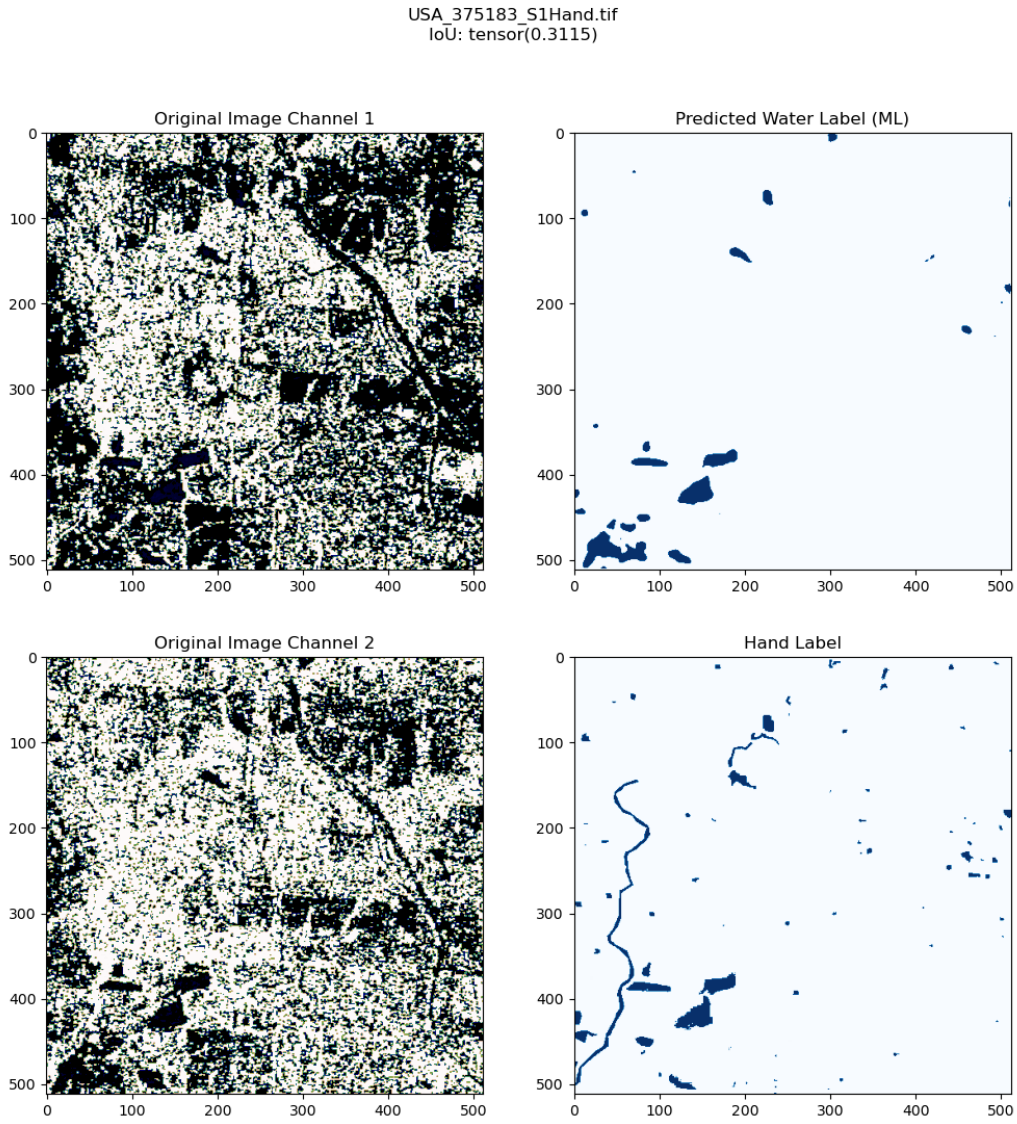

***Figure 3**. Prediction from the S1 Hand Labelled data subset. Note the loss of detail in the neural network produced imagery, particularly when it comes to channels and small pools.*

S1 Hand Trained Model

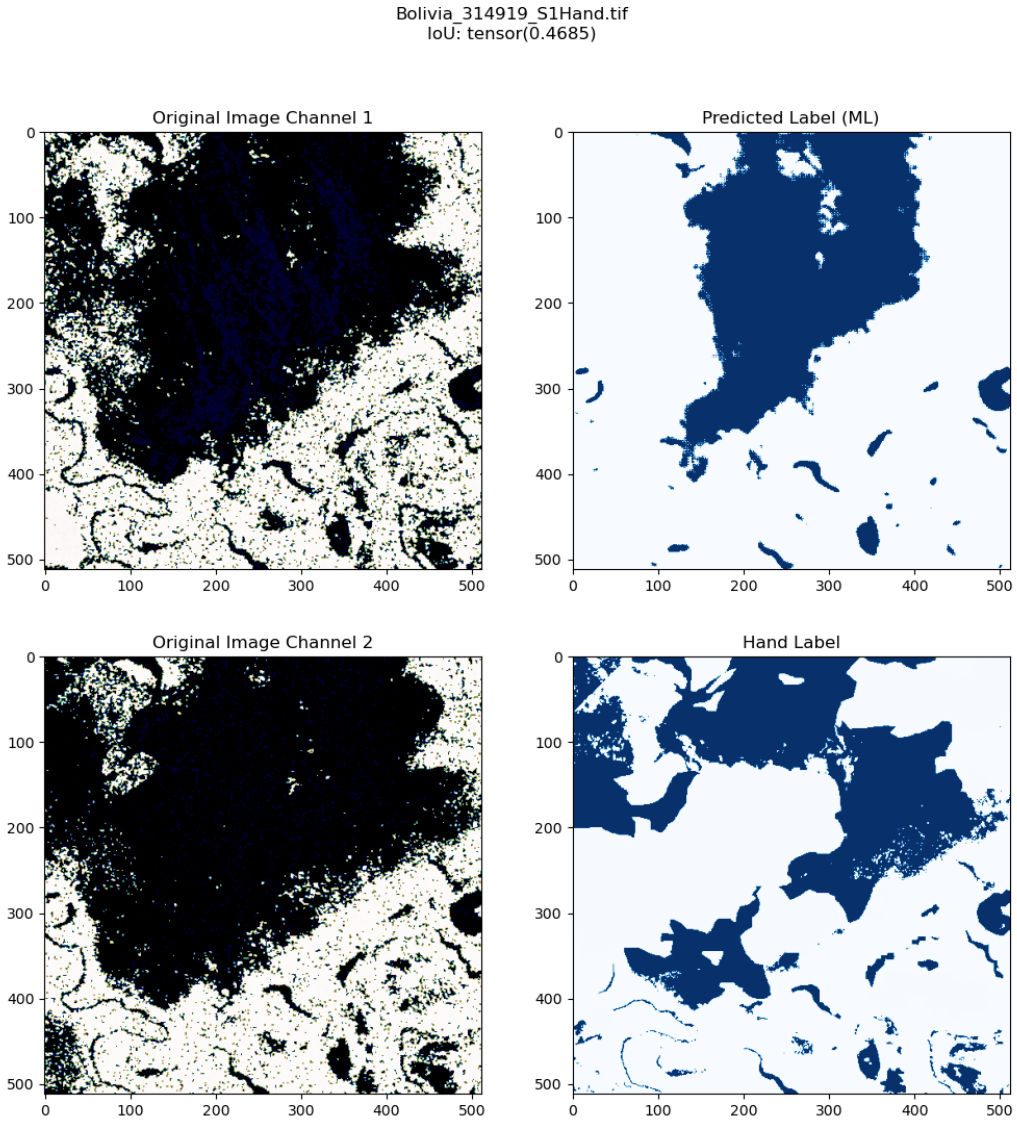

***Figure 4**. Prediction from the S1 Hand Labelled data subset using the model trained on the same data. While the IoU is moderate at 0.4685, the labels seem to show almost two different images.*

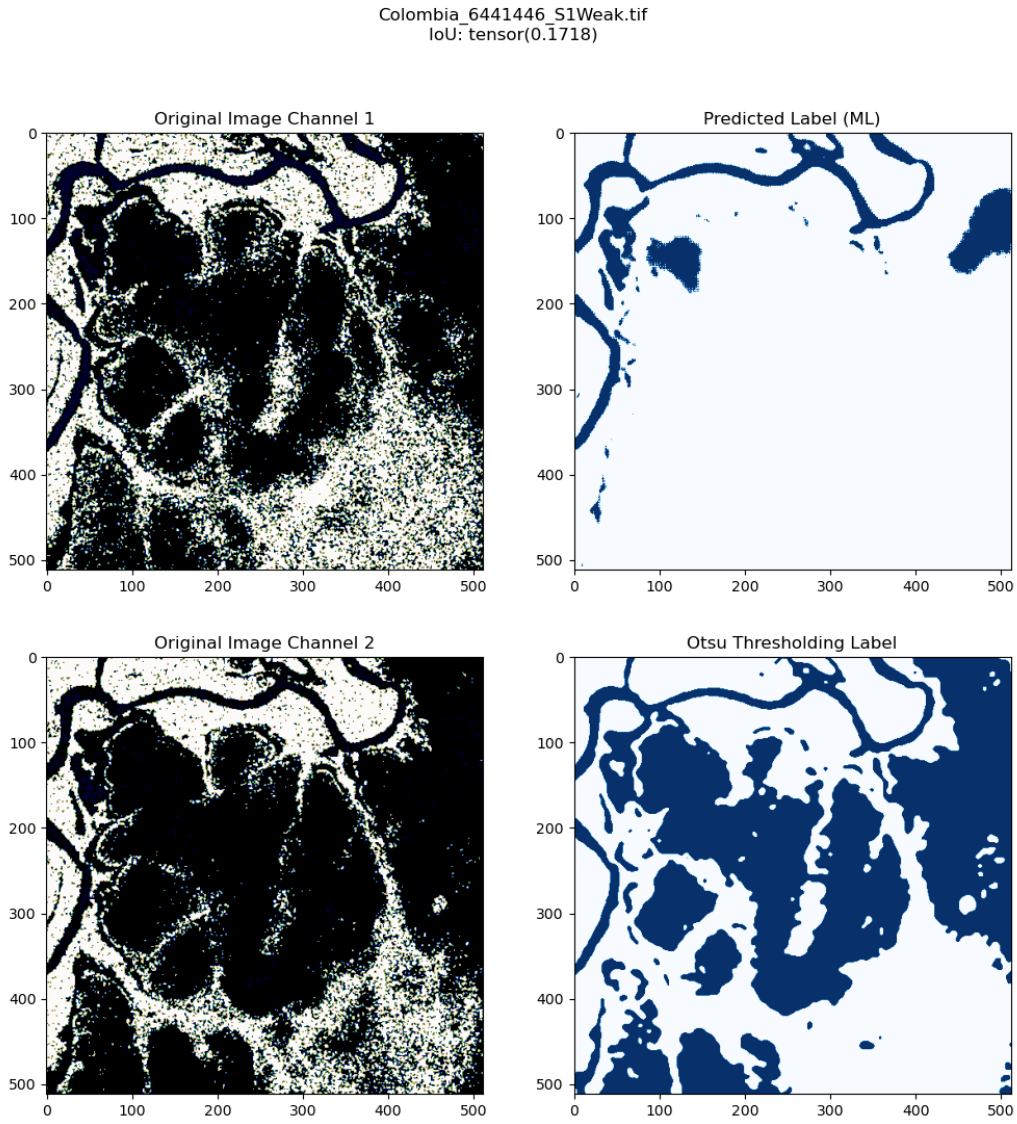

Figure 5. *Using model trained on the small hand labelled data set generally produces poor results. In this graphic, the upper right image has been produced by the model trained on S1 Hand labelled data, where the lower right image has been produced by Otsu Thresholding.*

S2 Weak Trained Model

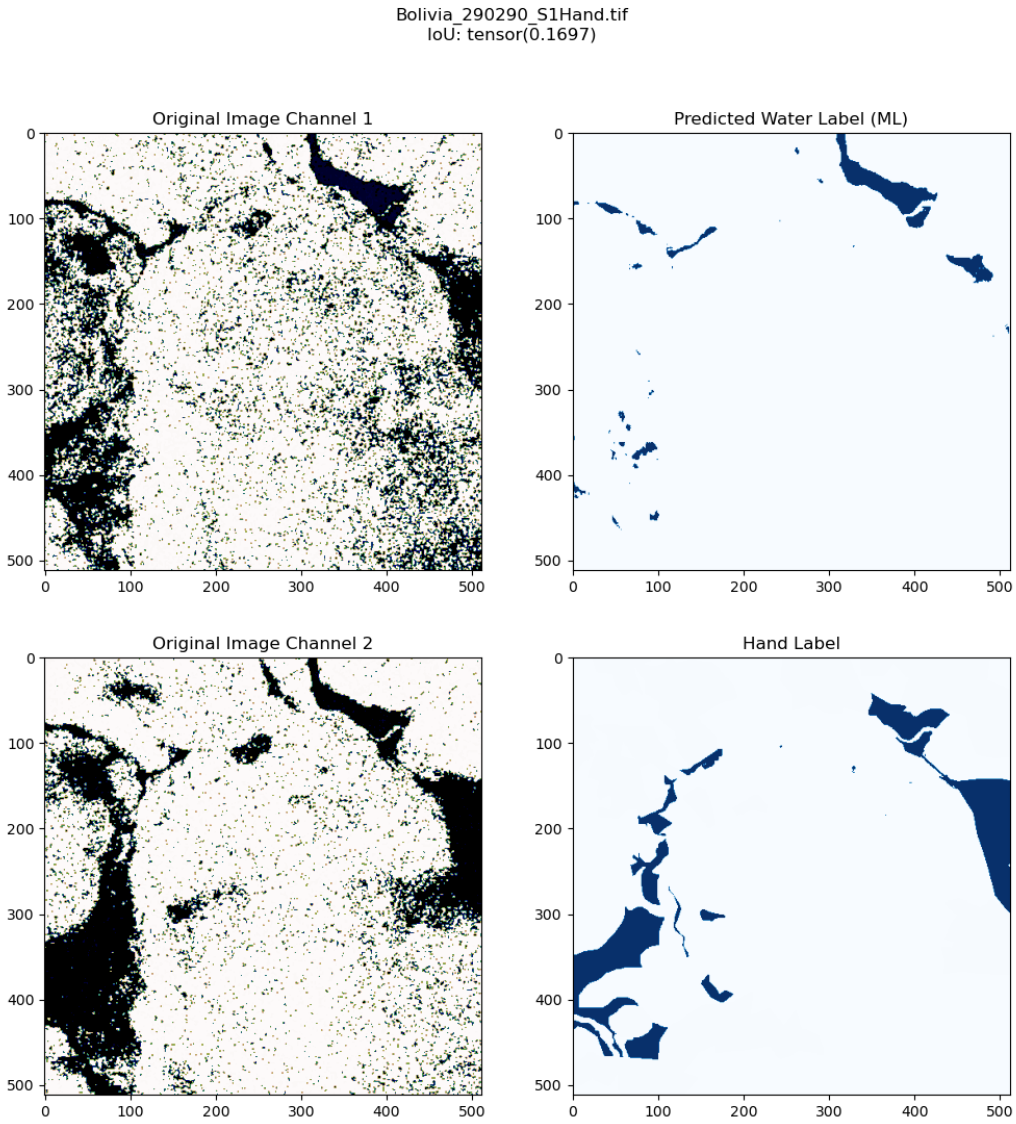

Figure 5. *This graphic shows the output from neural network trained on S2 weakly labelled data. The input is the same as Figure 1. There is notable loss of detail.*

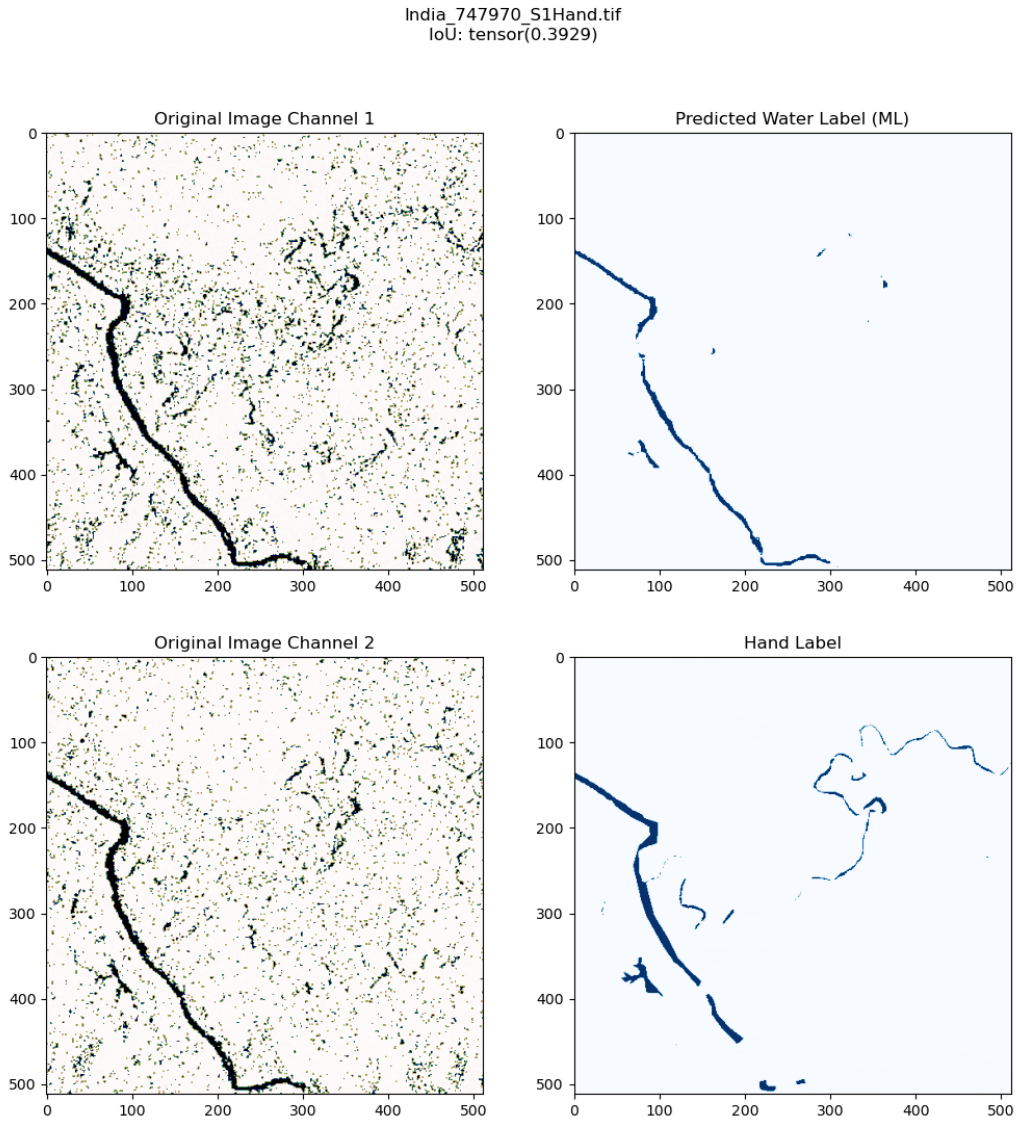

Figure 6. *This graphic shows the output from neural network trained on S2 weakly labelled data. While it is able to capture the larger channel, the smaller, more sinuous channel goes largely unnoticed by the neural network.*