# DATS 6203 Report

## Image Segmentation and Feature Detection with U-Net

### Introduction

Image classification, segmentation, and feature detection are extremely important machine learning applications used in many fields, including medical imaging,[[1]](#footnote-1) land use,[[2]](#footnote-2) and general object detection.[[3]](#footnote-3) Developed in 2015, the U-Net is a fully convolutional network (FCN) that performs image segmentation very well.[[4]](#footnote-4) As a fully convolutional network, the U-Net is able to segment images on a limited set of annotated data and retain that information as it relates to the original image.

In 2016, the United Kingdom’s Defence Science and Technology Laboratory (DSTL) created a Kaggle competition challenging participants to classify features in satellite images.[[5]](#footnote-5) The dataset contained labeling images with up to ten different features:

1. Buildings - large building, residential, non-residential, fuel storage facility, fortified building
2. Misc. Manmade structures
3. Road
4. Track - poor/dirt/cart track, footpath/trail
5. Trees - woodland, hedgerows, groups of trees, standalone trees
6. Crops - contour ploughing/cropland, grain (wheat) crops, row (potatoes, turnips) crops
7. Waterway
8. Standing water
9. Vehicle Large - large vehicle (e.g. lorry, truck, bus), logistics vehicle
10. Vehicle Small - small vehicle (car, van), motorbike

While the competition is not new, it does contain data that offers an opportunity to classify specific features in the images. The high number of different features also allows us to build models that can distinguish between similar features.

In this report, we will present results from training a U-Net on this dataset with the goal of distinguishing between five sets of features through binary classification:

1. Buildings and Misc. Manmade Structures
2. Roads and Tracks
3. Trees and Crops
4. Waterways and Standing Water
5. Large and Small Vehicles

### Description of the Dataset

The Kaggle dataset contains 150 satellite images that contain up to 10 different features saved as .tiff files. Each file is very large (over 3000 x 3000 pixels) and the images are captured in 3-band red-blue-green and 16-band formats. The 16 band images contain wavelengths from outside the visible spectrum.

The images are also accompanied by a listing of features and coordinates for the polygons that surround a labeled feature including buildings, miscellaneous manmade structures, roads, tracks, trees, crops, waterways, standing water, large vehicles, and small vehicles. These labels are supplied in two different formats: geojson files and well-known text (wkt) representations in a .csv file.

An example of a raw image is included below:

A close up of a plant

Description automatically generated

*Figure 1 – Example image from the Kaggle DSTL dataset saved as a .png*

### Neural Network and Training Algorithm

For this project, a U-Net was used to perform image segmentation and feature detection. Many helpful resources were included in the notebooks and discussion section of the Kaggle competition that served as a jumping off point for achieving our goals of distinguishing similar features from the dataset.

The U-Net was chosen for two primary reasons:

1. It is designed for small numbers of annotated input data and significant levels of data augmentation.[[6]](#footnote-6)
2. Existing image classification and feature detection entries in the competition and other published research using this dataset found good results with the U-Net architecture.[[7]](#footnote-7)

The extremely large files sizes for the input images necessitated the augmentation of the dataset so it could be fed into a U-Net in a timely manner. Additionally, there were not that many images for training, so a U-Net was a good choice with these limitations.

A U-Net is an FCN built for image segmentation through its architecture. U-Nets perform convolutional operations on the data, called down sampling because each convolution operation reduces the size of the image. It then up samples the filtered data back into an output segmentation map.[[8]](#footnote-8)

The U-Net performs this first step, known as the contraction path, which is a feed forward combination of convolutional and max pooling layers. It then performs a ‘symmetric expanding path’ which up samples the data into a mapping of predicted classes that is the same size of the original image. This mapping data can be visualized as a ‘mask’ over the image indicating where a feature is located.

Using the Generalized Neural Network notation,[[9]](#footnote-9) a U-Net can be generally described in a few steps:

3



3



T

*Figure 2 – Generalized Network Representation of a U-Net. The convolution in layer 3 is transposed to up sample the mapped output to the original image size.*

The network first convolves over the input data and then pools the output through layers 1 and 2 in the diagram. Next the pooling operates in ‘reverse’ in layer 3, where the pooled data is resized sequentially to the same dimensions as the original input, sometimes with the help of some padding. Like the kernel of a convolution layer, the transposed convolutions that up sample the inputs are learned.[[10]](#footnote-10)

These layers can be duplicated any number of times; however, every down sampling layer should have a matching up sampling layer, so the model achieves its namesake symmetric ‘U’ architecture.

For our model, we used the following code to build the Keras U-Net and it was modeled on some existing examples in the Kaggle competition’s notebook section.[[11]](#footnote-11) The model has 10 layers and incorporates normalization, Exponential Linear Unit (ELU) activations, pooling, and cropping to achieve better performance.

Example of convolution and pooling layer:

def get\_unet0():

    inputs = keras.Input((img\_rows, img\_cols, num\_channels))

    conv1 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(inputs)

    conv1 = BatchNormalization()(conv1)

    conv1 = keras.layers.advanced\_activations.ELU()(conv1)

    conv1 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv1)

    conv1 = BatchNormalization()(conv1)

    conv1 = keras.layers.advanced\_activations.ELU()(conv1)

    pool1 = MaxPooling2D(pool\_size=(2, 2))(conv1)

Example of up sampling layer:

up9 = concatenate([UpSampling2D(size=(2, 2))(conv8), conv1], axis=3)

    conv9 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up9)

    conv9 = BatchNormalization()(conv9)

    conv9 = keras.layers.advanced\_activations.ELU()(conv9)

    conv9 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv9)

    crop9 = Cropping2D(cropping=((16, 16), (16, 16)))(conv9)

    conv9 = BatchNormalization()(crop9)

    conv9 = keras.layers.advanced\_activations.ELU()(conv9)

    conv10 = Conv2D(num\_mask\_channels, (1, 1), activation='sigmoid')(conv9)

We also tried building a U-Net model in Pytorch, but due to time constraints were not able to fully operationalize this version of the U-Net.

Example of U-Net in Pytorch:

        self.dconv\_down0 = double\_conv(3, 32)

        self.dconv\_down1 = double\_conv(32, 64)

        self.dconv\_down2 = double\_conv(64, 128)

        self.dconv\_down3 = double\_conv(128, 256)

        self.dconv\_down4 = double\_conv(256, 512)

        self.maxpool = nn.MaxPool2d(2)

        self.upsample = nn.Upsample(scale\_factor=2, mode='bilinear', align\_corners=True)

        self.dconv\_up3 = double\_conv(256 + 512, 256)

        self.dconv\_up2 = double\_conv(128 + 256, 128)

        self.dconv\_up1 = double\_conv(128 + 64, 64)

        self.dconv\_up0 = double\_conv(64 + 32, 32)

        self.conv\_last = nn.Conv2d(32, n\_class, 1)

For the training algorithm, we followed a few steps:

1. Use a data generator to create batches of randomly cropped images from our original input data and feed these smaller images into the model.
2. Augment the data by randomly rotating and flipping the data to provide additional training images to the model. After training on certain epochs, Hard Example Mining was used to generate data and feed the model with samples that it did not perform well on.
3. Monitor binary cross-entropy (BCE) loss and our performance index for model performance and to catch issues like overfitting.

### Experimental Setup

The data used to train and test the network provided some unique challenges and opportunities for problem-solving. The files were very large and there were not many of them, so data augmentation was a key component of improving model performance. We also needed to stratify the data so it would help us to achieve our goal of building a network that can distinguish between two similar features.

The data was first split into cached sets of images associated with the five groups of similar features. 16-band and 3-band raw files were converted into 22 channel images that were optimized for distinguishing certain features based on infrared spectral properties.[[12]](#footnote-12)

Code to read raw images:

def read\_image\_22(image\_id):

    img\_a = np.transpose(tiff.imread(data\_path + "/sixteen\_band/{}\_A.tif".format(image\_id)), (1, 2, 0))

    img\_m = np.transpose(tiff.imread(data\_path + "/sixteen\_band/{}\_M.tif".format(image\_id)), (1, 2, 0)) # h w c

    img\_3 = np.transpose(tiff.imread(data\_path + "/three\_band/{}.tif".format(image\_id)), (1, 2, 0))

    img\_p = tiff.imread(data\_path + "/sixteen\_band/{}\_P.tif".format(image\_id)).astype(np.float32)

    height, width, \_ = img\_3.shape

    rescaled\_M = cv2.resize(img\_m, (width, height), interpolation=cv2.INTER\_CUBIC)

    rescaled\_A = cv2.resize(img\_a, (width, height), interpolation=cv2.INTER\_CUBIC)

    rescaled\_P = cv2.resize(img\_p, (width, height), interpolation=cv2.INTER\_CUBIC)

    rescaled\_P = np.expand\_dims(rescaled\_P, 2)

    stretched\_A = stretch\_n(rescaled\_A)

    rescaled\_M = stretch\_n(rescaled\_M)

    rescaled\_P = stretch\_n(rescaled\_P)

    img\_3 = stretch\_n(img\_3)

    aligned\_A = \_align\_two\_rasters(img\_3, stretched\_A, 'A')

    rescaled\_M = \_align\_two\_rasters(img\_3, rescaled\_M, 'M')

    rescaled\_P = \_align\_two\_rasters(img\_3, rescaled\_P, 'P')

    rescaled\_P = np.expand\_dims(rescaled\_P, 2)

    image\_r = img\_3[:, :, 0]

    image\_g = img\_3[:, :, 1]

    nir = rescaled\_M[:, :, 7]

    re = rescaled\_M[:, :, 5]

    ndwi = (image\_g - nir) / (image\_g + nir)

    ndwi = np.expand\_dims(ndwi, 2) # crop tree

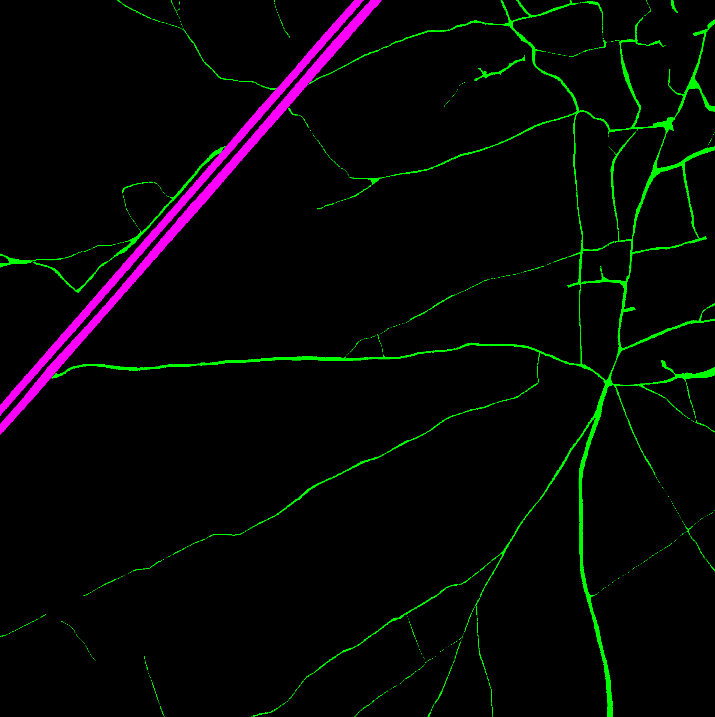
    ccci = (nir - re) / (nir + re) \* (nir - image\_r) / (nir + image\_r)

    ccci = np.expand\_dims(ccci, 2)

    result = np.concatenate([aligned\_A, rescaled\_M, rescaled\_P, ndwi, ccci, img\_3], axis=2)

The labels for these images were also masked over the original image to obtain our target image for the U-Net.

A picture containing rain, white, black, covered

Description automatically generated

*Figure 3 – Example of raw image and its associated mask for the ‘roads and tracks’ model*

Full sets of the input images and the masks were saved as .h5 files for each of the five sets of similar features.

Each model has a data generator that would randomly crop and augment the data files to provide more examples for the U-Net to learn from. These augmented files were generated in tandem with the labeled masks so the network could update kernels that matched our target features. In effect, this served a similar purpose to k-fold cross validation. When using fit\_generator, the number of samples processed for each epoch is batch\_size \* steps\_per\_epochs. steps\_per\_epochs here is total number of steps (batches of samples) to yield from generator before declaring one epoch finished and starting the next epoch (specified in the samples\_per\_epoch parameter below). The biggest batch size we could choose, considering computation power, is 128. However, the raw image is 900 times bigger in size of the image we fed to the model ((3360 \* 3360) / (112 \* 112) = 30). With 100-400 steps\_per\_epochs, the model was trained on enough samples per epochs.

Example of the code for loading the data into the U-Net for distinguishing standing and flowing water

model.fit\_generator(generator=data\_generator

(X\_train, y\_train, batch\_size, horizontal\_flip=True, vertical\_flip=True, swap\_axis=True),

epochs=nb\_epoch, verbose=1, samples\_per\_epoch=batch\_size \* 100,

validation\_data=data\_generator

(X\_train, y\_train, 128, horizontal\_flip=False, vertical\_flip=False, swap\_axis=False),

validation\_steps = 4,

callbacks=[ModelCheckpoint(filepath, monitor="val\_loss", save\_best\_only=True, save\_weights\_only=True)],

workers=8)

After training on certain epoch, Hard Example Mining was used to generate data and train the model with samples that it is not performing well on. That is:

1. The model was validated on a batch of samples.
2. One half of samples with higher loss were selected and fed to next epoch, together with another half randomly generated samples.

@threadsafe\_generator  
def mine\_hard\_samples(model, datagen, batch\_size):  
 while True:  
 samples, targets, loss = [], [], []  
 x\_data, y\_data = next(datagen)  
 preds = model.predict(x\_data)  
 for i in range(len(preds)):  
 loss.append(K.mean(jaccard\_coef\_loss(y\_data[i], preds[i])))  
 ind = np.argpartition(np.asarray(loss), -int(batch\_size / 2))[-int(batch\_size / 2):]  
 samples += x\_data[ind].tolist()  
 targets += y\_data[ind].tolist()  
  
 x\_data, y\_data = next(datagen)  
 samples += x\_data[:int(batch\_size/2)].tolist()  
 targets += y\_data[:int(batch\_size/2)].tolist()  
 samples, targets = map(np.array, (samples, targets))

We also initially explored the idea of building the network in both Keras and Pytorch to compare the performance. Due to time constraints and issues with Pytorch implementation (including very limited resources to help with building a U-Net in Pytorch), our experiment was designed with Keras in mind.

To gauge the performance of our models, we used the Jaccard Index (also known as the Intersection over Union) to measure the statistical similarity of sample sets from training and test data. The Jaccard Index measure the area of overlap between the true value (in our case a feature label’s associated polygon coordinates) and predicted value (in our case the bounded area of a predicted feature). This area is then divided by the total area of both true and predicted values.[[13]](#footnote-13)



*Figure 4 – Visual representation of the Jaccard Index. Image from* [*DeepAI.org*](https://deepai.org/machine-learning-glossary-and-terms/jaccard-index)*.*

The model for each set of features minimized this Jaccard Index value and included Binary Cross-Entropy loss to help gauge its performance against a validation set to minimize overfitting..

Model parameters were guided by previous research and examples from the original Kaggle competition. Each of the five models we ran were tuned according to the validation results to maximize performance and minimize overfitting. Generally, we found that high numbers of training epochs led to overfitting and poor performance on validation data. Conversely, increasing the numbers of samples in each epoch through from the data augmentation steps dramatically improved performance.

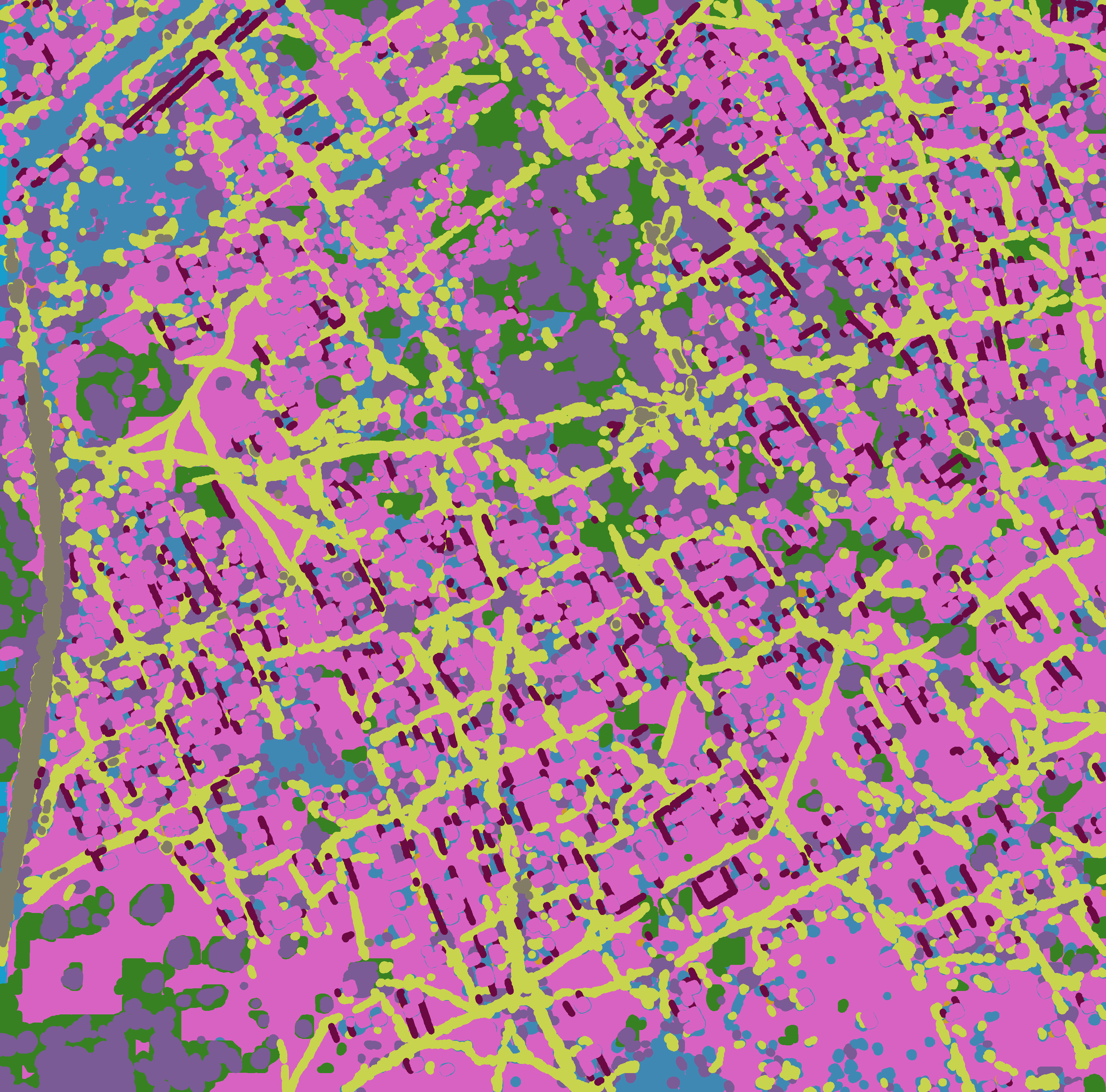
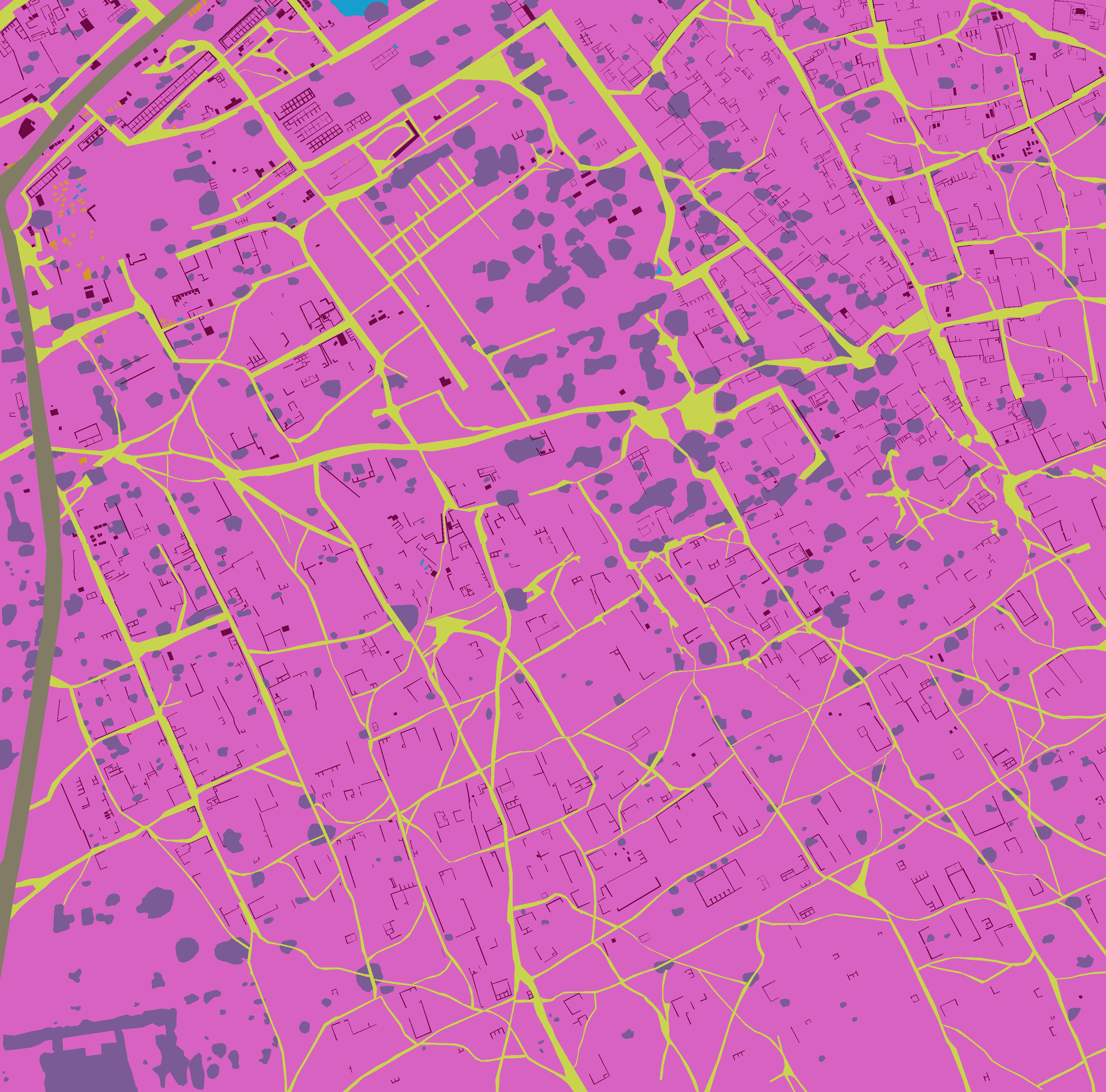
### Results

The results of our modeling seem to be consistent with other work done on this dataset. Distinguishing between two similar features is highly dependent on the characteristics of that feature.

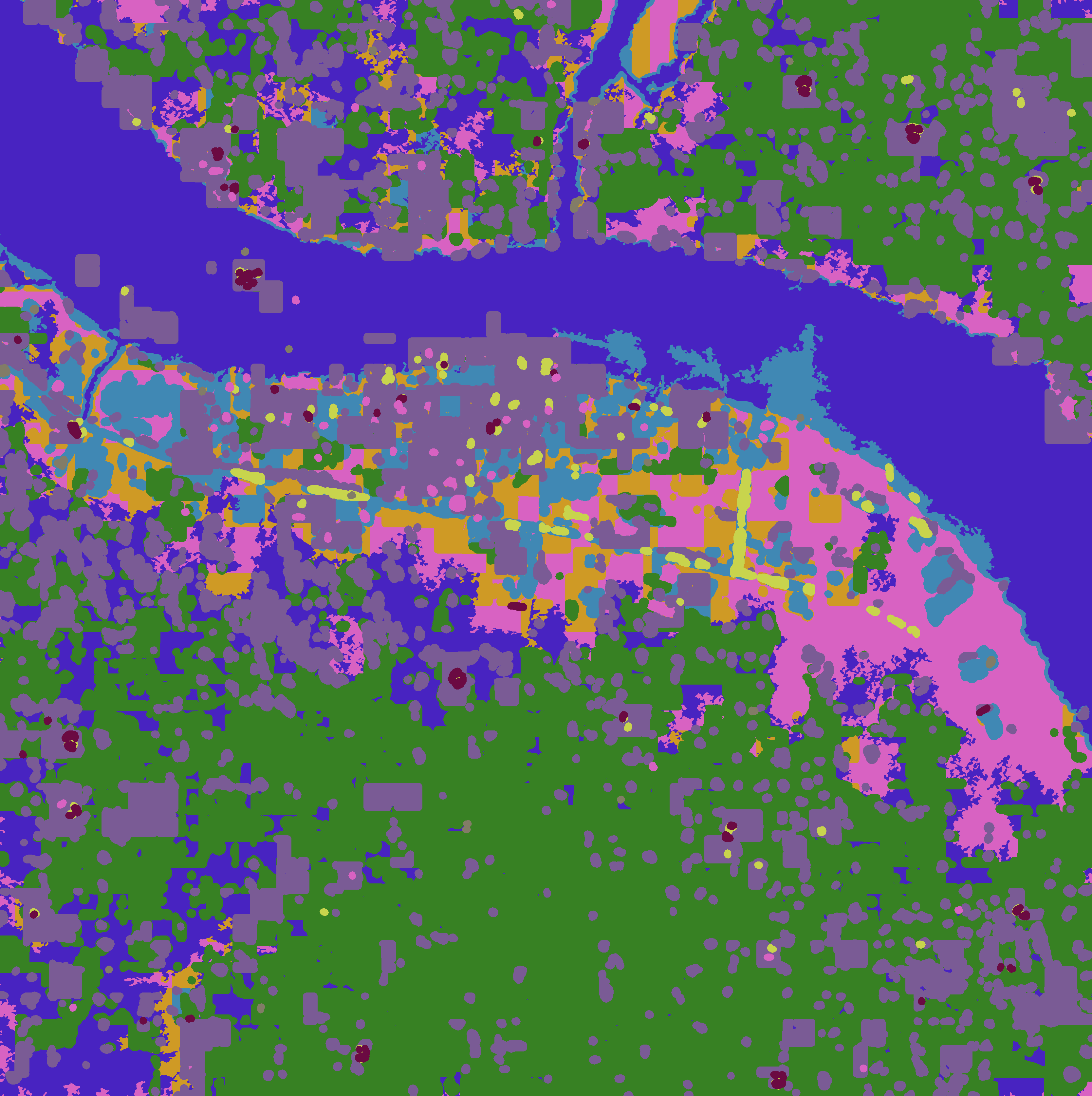
|  |  |  |
| --- | --- | --- |
| **Type of Feature Detection** | **Training Jaccard Index** | **Test Jaccard Index** |
| Roads and Tracks | 0.47 | 0.40 |
| Buildings and Misc. Manmade Structures | 0.54 | 0.49 |
| Trees and Crops | 0.55 | 0.48 |
| Large and Small Vehicles | 0.26 | 0.11 |
| Standing and Moving Water | 0.21 | 0.04 |

*Table 1 – Summary of each U-Net’s performance for distinguishing between two similar features.*

Our models performed well when detecting the differences between roads and tracks, buildings and miscellaneous manmade structures, and trees and crops. However, performance diminished when detecting the difference between large and small vehicles and standing and moving water.



*Figure 5 - Example Raw image of 6100\_2\_3 in the training set (left), mask generated from labeled polygon (middle) and segmented image predicted (right)*

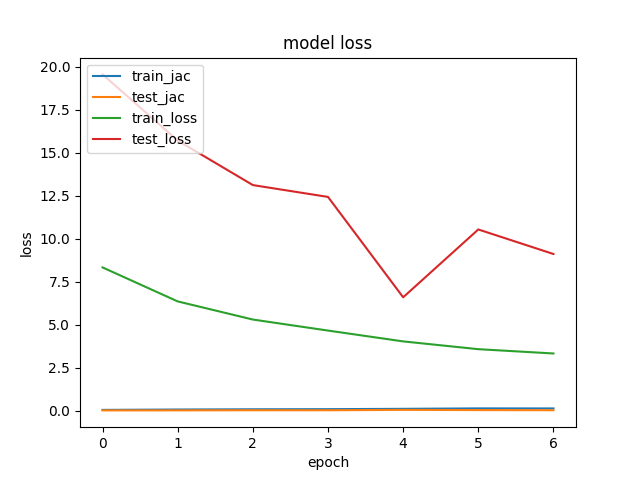
*Figure 6 - Example Raw image of 6050\_4\_4 in the test set (left), and segmented image predicted (right)*

Existing research on this dataset found very similar results.[[14]](#footnote-14) For example, vehicles are likely too small to be segmented precisely on satellite images compared to other classes such as buildings and crop fields.

We can also review the metrics from our models to see how consistent and accurate they are at detecting features and minimizing loss.

More details about the performance of our models can be found below. As can be seen, training loss decreases with each epoch signaling good convergence.

#### Roads and Tracks



*Figure 7 – Graph of model loss at each epoch for training and validation data*

#### Buildings and Misc. Manmade Structures



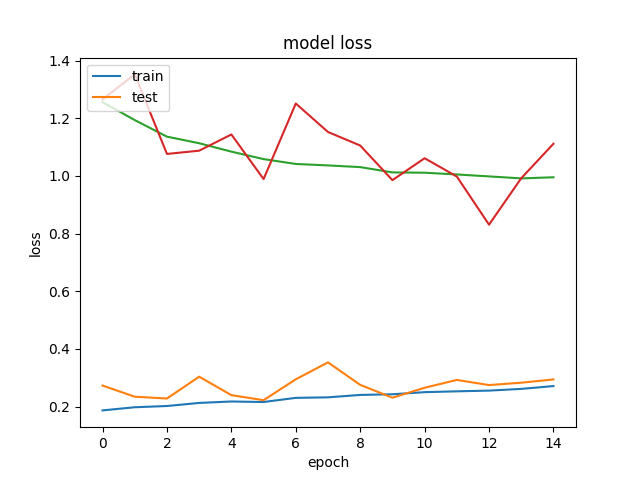
Train\_loss

Val\_BCE

Train\_BCE

Val\_loss

*Figure 8 - Graph of model loss at each epoch for training and validation data*



Val\_BCE

Train\_BCE

Train\_loss

Val\_loss

*Figure 9 – Example of decreasing Jaccard Index performance with additional epochs*

After certain epochs, train\_BCE kept decreasing and test\_BCE fluctuate, while train\_loss = (-log(jac) + BCE) increase, which means that the Jaccard Index score increased.

#### Trees and Crops

The model performed reasonably well for distinguishing between trees and crops from the dataset. While training, the number of epochs were kept low to minimize overfitting

A screenshot of a cell phone

Description automatically generated

Train\_Jaccard

Train\_BCE

Val\_Jaccard

Val\_BCE

*Figure 10 – Graph of model loss at each epoch for training and validation data*

A close up of a map

Description automatically generated

Increasing BCE Indicated Overfitting

*Figure 11 – Example of overfitting indicated by increasing BCE with each epoch*

#### Large and Small Vehicles

The model did not perform very well for distinguishing between large and small vehicles from the dataset. As with other training, epochs were kept low and a larger number of augmented samples were introduced to improve performance

A close up of a map

Description automatically generated

Val\_BCE

Val\_Jaccard

Train\_Jaccard

Train\_BCE

*Figure 12 – Graph of model loss at each epoch for training and validation data*

#### Standing and Moving Water

As with large and small vehicles, the model did not perform well for distinguishing between standing and moving water from the dataset. Once again, epochs were kept low and a larger number of augmented samples were introduced to improve performance

A picture containing screenshot

Description automatically generated

Val\_Jaccard

Train\_BCE

Train\_Jaccard

Val\_BCE

*Figure 13 – Graph of model loss at each epoch for training and validation data*

There were also some other creative solutions to aid in identifying water that do not rely on a neural network.[[15]](#footnote-15) The dataset contains enough spectra in the various bands to calculate the reflective index of each pixel in the image. Since water tends to have a consistent Canopy Chlorophyll Content Index (CCCI), or reflective index and is unique from other features, this CCCI can serve as a filter to mask over areas of water (indicated by CCCI threshold over 0.11). By using this CCCI, the Jaccard Index increases to ~0.5 on this data set according to previous research[[16]](#footnote-16) and could be helpful for further distinguishing water from other parts of this dataset or future datasets.

*Figure 13 – Example of a mask being applied to water (green) wherever CCCI exceeds 0.11*

Example code using the CCCI to distinguish water:

def mask2poly\_fastwater(predicted\_mask, x\_scaler, y\_scaler):  
 polygons = extra\_functions.mask2polygons\_layer(predicted\_mask, epsilon=0, min\_area=10000)  
 polygons = shapely.affinity.scale(polygons, xfact=1.0 / x\_scaler, yfact=1.0 / y\_scaler, origin=(0, 0, 0))  
 return shapely.wkt.dumps(polygons)

def mask2poly\_slowwater(predicted\_mask, x\_scaler, y\_scaler):  
 polygons = extra\_functions.mask2polygons\_layer(predicted\_mask, epsilon=0, min\_area=1000)  
  
 polygons = MultiPolygon([x for x in polygons if 270000 < x.area < 300000 or x.area < 90000])  
  
 polygons = shapely.affinity.scale(polygons, xfact=1.0 / x\_scaler, yfact=1.0 / y\_scaler, origin=(0, 0, 0))  
 return shapely.wkt.dumps(polygons)

image\_r = img\_3[:, :, 0]  
nir = rescaled\_M[:, :, 7]  
re = rescaled\_M[:, :, 5]  
  
ccci = (nir - re) / (nir + re) \* (nir - image\_r) / (nir + image\_r)

predicted\_mask = (ccci > 0.11).astype(np.float32)  
  
if predicted\_mask.sum() <= 500000:  
 result += [(image\_id, 7, 'MULTIPOLYGON EMPTY')]  
else:  
 result += [(image\_id, 7, mask2poly\_fastwater(predicted\_mask, x\_scaler, y\_scaler))]  
if predicted\_mask.sum() > 680000:  
 result += [(image\_id, 8, 'MULTIPOLYGON EMPTY')]  
else:  
 result += [(image\_id, 8, mask2poly\_slowwater(predicted\_mask, x\_scaler, y\_scaler))]

#### Additional Discussion

According to the performance of our models, we were happy with the general conclusions of this project. The Jaccard Index serves a unique role in quantifying the ability of our models to distinguish between similar features. Like a confusion matrix, it shows how well a FCN performs masking of the original image and using other metrics like BCE allowed us to understand how our models were working better than a traditional accuracy score or some other metric that is used for classification. This dataset and image segmentation process was also unique because we could visually see where images were correctly or incorrectly segmented.

### Summary and Conclusions

This project was very exciting to work on because of the amount that we were able to learn from it. The DSTL dataset posed some unique challenges, but a multitude of existing resources and a lot of iteration helped us to achieve our goal of building U-Nets that can distinguish between similar features in an image.

There were two significant limitations that we faced while working on this project: time and the data itself.

The most impactful limitation to this project was a lack of time. Because our goal was to distinguish between two similar features, we had to build many models. The large size of the data meant it took a very long time to run these models and perform our analysis, even with the help of GPU. Each model would take multiple hours to train and it limited our ability to tune these models effectively. Luckily, there were some very good resources to help alleviate some of these concerns (listed in our additional references section), but it was still a barrier to accomplishing our goals.

A surprising challenge was building models from the training data. Each picture was massive, but there were not that many unique images. This increased the likelihood of overfitting and reduced performance because subsamples of our training data were augmented and duplicated multiple times. Some features were also remarkably similar and hard to distinguish. As noted in the discussion of results, features like vehicles and water were very hard to distinguish and additional outside information or data could have helped with this problem.

Even with these challenges, the project was a very good opportunity to explore a new type of neural network. The U-Net architecture lent itself well to new techniques such as batching the images from our dataset or implementing new types of augmentation. This project could also be a good steppingstone for performing further research on this dataset. For example, it would be interesting to build a conglomerated model could learn even more from other objects nearby. A vehicle might increase the likelihood of a road being classified and vice-versa.

This project also showcased the power of testing out neural network architectures. While our scope was limited to a Keras implementation of a U-Net it is not hard to image what sort of tasks could be accomplished by branching out into a Pytorch implementation of a ResNet. Finally, the model building and training processes used in this project are not limited to this dataset. All the techniques and tricks are new skills we can use in future data science projects.

### References

Additional references that were helpful for this project and not footnoted earlier are included below.

#### Data Loading and Preprocessing

* Using the Kaggle API to download data: <https://gist.github.com/jayspeidell/d10b84b8d3da52df723beacc5b15cb27>
* Loading the large files and mitigating errors: <http://stackoverflow.com/questions/15063936/csv-error-field-larger-than-field-limit-131072>
* Help with fixing errors in .h5 files: <https://github.com/h5py/h5py/issues/441>
* Process masking with polygons and cv2: <http://docs.opencv.org/3.1.0/d9/d8b/tutorial_py_contours_hierarchy.html>

#### Model Building/Training

* Great end-to-end example with metrics: <https://www.kaggle.com/drn01z3/end-to-end-baseline-with-u-net-keras>
* Additional example of U-Net: <https://www.kaggle.com/ceperaang/lb-0-42-ultimate-full-solution-run-on-your-hw>

# Appendix

## Data Loading

#This dataloading is set up for the images from the DSTL dataset

import os

import numpy as np

import pandas as pd

#Guide to download Kaggle datasets directly found here: https://gist.github.com/jayspeidell/d10b84b8d3da52df723beacc5b15cb27

import kaggle

api\_token = {"username":"USERNAME\_GOES\_HERE","key":"KEY\_GOES\_HERE"}

import json

import zipfile

import os

with open('/root/.kaggle/kaggle.json', 'w') as file:

    json.dump(api\_token, file)

os.system('kaggle competitions download -c dstl-satellite-imagery-feature-detection')

if not os.path.exists("/content/competitions/dstl-satellite-imagery-feature-detection"):

    os.makedirs("/content/competitions/dstl-satellite-imagery-feature-detection")

os.chdir('/content/competitions/dstl-satellite-imagery-feature-detection')

for file in os.listdir():

  zip\_ref = zipfile.ZipFile(file, 'r')

  zip\_ref.close()

  os.system("unzip sixteen\_band.zip")

  os.system("unzip grid\_sizes.csv.zip")

  os.system("unzip sample\_submissions.csv.zip")

  os.system("unzip three\_band.zip")

  os.system("unzip train\_geojson\_v3.zip")

  os.system("unzip train\_wkt\_v4.zip")

## Preprocessing

from \_\_future\_\_ import division

from shapely.wkt import loads as wkt\_loads

import os

import shapely

import shapely.geometry

import shapely.affinity

import pandas as pd

from collections import defaultdict, OrderedDict

import csv

import sys

import cv2

from shapely.geometry import MultiPolygon, Polygon

import shapely.wkt

import shapely.affinity

import numpy as np

import tifffile as tiff

# dirty hacks from SO to allow loading of big cvs's

# without decrement loop it crashes with C error

# http://stackoverflow.com/questions/15063936/csv-error-field-larger-than-field-limit-131072

maxInt = sys.maxsize

decrement = True

while decrement:

    # decrease the maxInt value by factor 10

    # as long as the OverflowError occurs.

    decrement = False

    try:

        csv.field\_size\_limit(maxInt)

    except OverflowError:

        maxInt = int(maxInt/10)

        decrement = True

data\_path = os.getcwd()

train\_wkt = pd.read\_csv(os.path.join(data\_path, 'train\_wkt\_v4.csv'))

gs = pd.read\_csv(os.path.join(data\_path, 'grid\_sizes.csv'), names=['ImageId', 'Xmax', 'Ymin'], skiprows=1)

shapes = pd.read\_csv(os.path.join(data\_path, '3\_shapes.csv'))

epsilon = 1e-15

def get\_class\_image(classes):

    class\_image = defaultdict(list)

    selected = train\_wkt[train\_wkt['MultipolygonWKT'] != 'MULTIPOLYGON EMPTY']

    for i in range(len(selected)):

        class\_image[selected.iloc[i, 1]].append(selected.iloc[i, 0])

    class\_image = OrderedDict(sorted(class\_image.items()))

    imageIDs = set(class\_image[classes[0]] + class\_image[classes[1]])

    return imageIDs

def get\_scalers(height, width, x\_max, y\_min):

    """

    :param height:

    :param width:

    :param x\_max:

    :param y\_min:

    :return: (xscaler, yscaler)

    """

    w\_ = width \* (width / (width + 1))

    h\_ = height \* (height / (height + 1))

    return w\_ / x\_max, h\_ / y\_min

def polygons2mask\_layer(height, width, polygons, image\_id):

    """

    :param height:

    :param width:

    :param polygons:

    :return:

    """

    x\_max, y\_min = \_get\_xmax\_ymin(image\_id)

    x\_scaler, y\_scaler = get\_scalers(height, width, x\_max, y\_min)

    polygons = shapely.affinity.scale(polygons, xfact=x\_scaler, yfact=y\_scaler, origin=(0, 0, 0))

    img\_mask = np.zeros((height, width), np.uint8)

    if not polygons:

        return img\_mask

    int\_coords = lambda x: np.array(x).round().astype(np.int32)

    exteriors = [int\_coords(poly.exterior.coords) for poly in polygons]

    interiors = [int\_coords(pi.coords) for poly in polygons for pi in poly.interiors]

    cv2.fillPoly(img\_mask, exteriors, 1)

    cv2.fillPoly(img\_mask, interiors, 0)

    return img\_mask

def polygons2mask(height, width, polygons, image\_id):

    num\_channels = len(polygons)

    result = np.zeros((num\_channels, height, width))

    for mask\_channel in range(num\_channels):

        result[mask\_channel, :, :] = polygons2mask\_layer(height, width, polygons[mask\_channel], image\_id)

    return result

def generate\_mask(image\_id, height, width, start, num\_mask\_channels, train=train\_wkt):

    """

    :param image\_id:

    :param height:

    :param width:

    :param num\_mask\_channels: numbers of channels in the desired mask

    :param train: polygons with labels in the polygon format

    :return: mask corresponding to an image\_id of the desired height and width with desired number of channels

    """

    mask = np.zeros((num\_mask\_channels, height, width))

    for mask\_channel in range(num\_mask\_channels):

        poly = train.loc[(train['ImageId'] == image\_id)

                         & (train['ClassType'] == mask\_channel + start + 1), 'MultipolygonWKT'].values[0]

        polygons = shapely.wkt.loads(poly)

        mask[mask\_channel, :, :] = polygons2mask\_layer(height, width, polygons, image\_id)

    return mask

def mask2polygons\_layer(mask, epsilon=1.0, min\_area=10.0):

    # first, find contours with cv2: it's much faster than shapely

    contours, hierarchy = cv2.findContours(((mask == 1) \* 255).astype(np.uint8), cv2.RETR\_CCOMP, cv2.CHAIN\_APPROX\_TC89\_KCOS)

    # create approximate contours to have reasonable submission size

    if epsilon != 0:

        approx\_contours = simplify\_contours(contours, epsilon)

    else:

        approx\_contours = contours

    if not approx\_contours:

        return MultiPolygon()

    all\_polygons = find\_child\_parent(hierarchy, approx\_contours, min\_area)

    # approximating polygons might have created invalid ones, fix them

    all\_polygons = MultiPolygon(all\_polygons)

    all\_polygons = fix\_invalid\_polygons(all\_polygons)

    return all\_polygons

def find\_child\_parent(hierarchy, approx\_contours, min\_area):

    # now messy stuff to associate parent and child contours

    cnt\_children = defaultdict(list)

    child\_contours = set()

    assert hierarchy.shape[0] == 1

    # http://docs.opencv.org/3.1.0/d9/d8b/tutorial\_py\_contours\_hierarchy.html

    for idx, (\_, \_, \_, parent\_idx) in enumerate(hierarchy[0]):

        if parent\_idx != -1:

            child\_contours.add(idx)

            cnt\_children[parent\_idx].append(approx\_contours[idx])

    # create actual polygons filtering by area (removes artifacts)

    all\_polygons = []

    for idx, cnt in enumerate(approx\_contours):

        if idx not in child\_contours and cv2.contourArea(cnt) >= min\_area:

            assert cnt.shape[1] == 1

            holes = [c[:, 0, :] for c in cnt\_children.get(idx, []) if cv2.contourArea(c) >= min\_area]

            contour = cnt[:, 0, :]

            poly = Polygon(shell=contour, holes=holes)

            if poly.area >= min\_area:

                all\_polygons.append(poly)

    return all\_polygons

def simplify\_contours(contours, epsilon):

    return [cv2.approxPolyDP(cnt, epsilon, True) for cnt in contours]

def fix\_invalid\_polygons(all\_polygons):

    if not all\_polygons.is\_valid:

        all\_polygons = all\_polygons.buffer(0)

        # Sometimes buffer() converts a simple Multipolygon to just a Polygon,

        # need to keep it a Multi throughout

        if all\_polygons.type == 'Polygon':

            all\_polygons = MultiPolygon([all\_polygons])

    return all\_polygons

def \_get\_xmax\_ymin(image\_id):

    xmax, ymin = gs[gs['ImageId'] == image\_id].iloc[0, 1:].astype(float)

    return xmax, ymin

def get\_shape(image\_id, band=3):

    if band == 3:

        height = shapes.loc[shapes['image\_id'] == image\_id, 'height'].values[0]

        width = shapes.loc[shapes['image\_id'] == image\_id, 'width'].values[0]

        return height, width

def stretch\_n(bands, lower\_percent=5, higher\_percent=95):

    out = np.zeros\_like(bands).astype(np.float32)

    n = bands.shape[2]

    for i in range(n):

        a = 0

        b = 1

        c = np.percentile(bands[:, :, i], lower\_percent)

        d = np.percentile(bands[:, :, i], higher\_percent)

        t = a + (bands[:, :, i] - c) \* (b - a) / (d - c)

        t[t < a] = a

        t[t > b] = b

        out[:, :, i] = t

    return out.astype(np.float32)

def \_align\_two\_rasters(img1,img2, band):

    i=0

    if band == 'A':

        i= 3

    elif band == 'M':

        i = 5

    p1 = img1[:, :, 1]

    p2 = img2[:, :, i]

    warp\_mode = cv2.MOTION\_EUCLIDEAN

    warp\_matrix = np.eye(2, 3, dtype=np.float32)

    criteria = (cv2.TERM\_CRITERIA\_EPS | cv2.TERM\_CRITERIA\_COUNT, 1000, 1e-7)

    (cc, warp\_matrix) = cv2.findTransformECC (p1, p2,warp\_matrix, warp\_mode, criteria, None, 1)

    img3 = cv2.warpAffine(img2, warp\_matrix, (img1.shape[1], img1.shape[0]), flags=cv2.INTER\_LINEAR + cv2.WARP\_INVERSE\_MAP)

    img3[img3 == 0] = np.average(img3)

    return img3

def read\_image\_22(image\_id):

    img\_a = np.transpose(tiff.imread(data\_path + "/sixteen\_band/{}\_A.tif".format(image\_id)), (1, 2, 0))

    img\_m = np.transpose(tiff.imread(data\_path + "/sixteen\_band/{}\_M.tif".format(image\_id)), (1, 2, 0)) # h w c

    img\_3 = np.transpose(tiff.imread(data\_path + "/three\_band/{}.tif".format(image\_id)), (1, 2, 0))

    img\_p = tiff.imread(data\_path + "/sixteen\_band/{}\_P.tif".format(image\_id)).astype(np.float32)

    height, width, \_ = img\_3.shape

    rescaled\_M = cv2.resize(img\_m, (width, height), interpolation=cv2.INTER\_CUBIC)

    rescaled\_A = cv2.resize(img\_a, (width, height), interpolation=cv2.INTER\_CUBIC)

    rescaled\_P = cv2.resize(img\_p, (width, height), interpolation=cv2.INTER\_CUBIC)

    rescaled\_P = np.expand\_dims(rescaled\_P, 2)

    stretched\_A = stretch\_n(rescaled\_A)

    rescaled\_M = stretch\_n(rescaled\_M)

    rescaled\_P = stretch\_n(rescaled\_P)

    img\_3 = stretch\_n(img\_3)

    aligned\_A = \_align\_two\_rasters(img\_3, stretched\_A, 'A')

    rescaled\_M = \_align\_two\_rasters(img\_3, rescaled\_M, 'M')

    rescaled\_P = \_align\_two\_rasters(img\_3, rescaled\_P, 'P')

    rescaled\_P = np.expand\_dims(rescaled\_P, 2)

    image\_r = img\_3[:, :, 0]

    image\_g = img\_3[:, :, 1]

    nir = rescaled\_M[:, :, 7]

    re = rescaled\_M[:, :, 5]

    ndwi = (image\_g - nir) / (image\_g + nir)

    ndwi = np.expand\_dims(ndwi, 2) # crop tree

    ccci = (nir - re) / (nir + re) \* (nir - image\_r) / (nir + image\_r)

    ccci = np.expand\_dims(ccci, 2)

    result = np.concatenate([aligned\_A, rescaled\_M, rescaled\_P, ndwi, ccci, img\_3], axis=2)

    # A = [:8], M = [8:16], P = [16], ndwi = [17], ccci = [18], 3 = [19:]

    '''

    SWIR (1195-2365 nm). This band cover different slices of the shortwave infrared. They are particularly useful for telling

    wet earth from dry earth, and for geology: rocks and soils that look similar in other bands often have strong contrasts in

    this band.

    NIR (772-954 nm).This band measures the near infrared. This part of the spectrum is especially important for ecology

    purposes because healthy plants reflect it. Information from this band is important for major reflectance indexes, such as

    NDWI.

    '''

    return result.astype(np.float32)

def make\_prediction\_cropped(model, X\_train, initial\_size=(572, 572), final\_size=(388, 388), num\_channels=22, num\_masks=2):

    shift = int((initial\_size[0] - final\_size[0]) / 2)

    height = X\_train.shape[1]

    width = X\_train.shape[2]

    if height % final\_size[1] == 0:

        num\_h\_tiles = int(height / final\_size[1])

    else:

        num\_h\_tiles = int(height / final\_size[1]) + 1

    if width % final\_size[1] == 0:

        num\_w\_tiles = int(width / final\_size[1])

    else:

        num\_w\_tiles = int(width / final\_size[1]) + 1

    rounded\_height = num\_h\_tiles \* final\_size[0]

    rounded\_width = num\_w\_tiles \* final\_size[0]

    padded\_height = rounded\_height + 2 \* shift

    padded\_width = rounded\_width + 2 \* shift

    padded = np.zeros((num\_channels, padded\_height, padded\_width))

    padded[:, shift:shift + height, shift: shift + width] = X\_train

    # add mirror reflections to the padded areas

    up = padded[:, shift:2 \* shift, shift:-shift][:, ::-1]

    padded[:, :shift, shift:-shift] = up

    lag = padded.shape[1] - height - shift

    bottom = padded[:, height + shift - lag:shift + height, shift:-shift][:, ::-1]

    padded[:, height + shift:, shift:-shift] = bottom

    left = padded[:, :, shift:2 \* shift][:, :, ::-1]

    padded[:, :, :shift] = left

    lag = padded.shape[2] - width - shift

    right = padded[:, :, width + shift - lag:shift + width][:, :, ::-1]

    padded[:, :, width + shift:] = right

    h\_start = range(0, padded\_height, final\_size[0])[:-1]

    assert len(h\_start) == num\_h\_tiles

    w\_start = range(0, padded\_width, final\_size[0])[:-1]

    assert len(w\_start) == num\_w\_tiles

    temp = []

    for h in h\_start:

        for w in w\_start:

            temp += [padded[:, h:h + initial\_size[0], w:w + initial\_size[0]]]

    prediction = model.predict(np.array(temp))

    predicted\_mask = np.zeros((num\_masks, rounded\_height, rounded\_width))

    for j\_h, h in enumerate(h\_start):

         for j\_w, w in enumerate(w\_start):

             i = len(w\_start) \* j\_h + j\_w

             predicted\_mask[:, h: h + final\_size[0], w: w + final\_size[0]] = prediction[i]

    return predicted\_mask[:, :height, :width]

"""

Script that scans 3 band tiff files and creates csv file with columns:

image\_id, width, height

"""

from \_\_future\_\_ import division

import tifffile as tiff

import os

from tqdm import tqdm

import pandas as pd

data\_path = os.getcwd()

three\_band\_path = os.path.join(data\_path, 'three\_band')

file\_names = []

widths\_3 = []

heights\_3 = []

for file\_name in tqdm(sorted(os.listdir(three\_band\_path))):

    # TODO: crashes if there anything except tiff files in folder (for ex, QGIS creates a lot of aux files)

    image\_id = file\_name.split('.')

    image\_3 = tiff.imread(os.path.join(three\_band\_path, file\_name))

    file\_names += [file\_name]

    \_, height\_3, width\_3 = image\_3.shape

    widths\_3 += [width\_3]

    heights\_3 += [height\_3]

df = pd.DataFrame({'file\_name': file\_names, 'width': widths\_3, 'height': heights\_3})

df['image\_id'] = df['file\_name'].apply(lambda x: x.split('.')[0])

df.to\_csv(os.path.join(data\_path, '3\_shapes.csv'), index=False)

import os

import pandas as pd

from collections import defaultdict

from collections import OrderedDict

import csv

data\_path = os.getcwd()

train\_wkt = pd.read\_csv(os.path.join(data\_path, 'train\_wkt\_v4.csv'))

class\_image = defaultdict(list)

selected = train\_wkt[train\_wkt['MultipolygonWKT'] != 'MULTIPOLYGON EMPTY']

for i in range(len(selected)):

    class\_image[selected.iloc[i, 1]].append(selected.iloc[i, 0])

class\_image = OrderedDict(sorted(class\_image.items()))

with open('class\_image.csv', 'w', newline="") as csv\_file:

    writer = csv.writer(csv\_file)

    for key, value in class\_image.items():

       writer.writerow([key, len(value), value])

### Cache data for training

"""

Script that caches train data for future training

"""

from \_\_future\_\_ import division

import os

import pandas as pd

import extra\_functions

import h5py

import numpy as np

import cv2

data\_path = os.getcwd()

train\_wkt = pd.read\_csv(os.path.join(data\_path, 'train\_wkt\_v4.csv'))

gs = pd.read\_csv(os.path.join(data\_path, 'grid\_sizes.csv'), names=['ImageId', 'Xmax', 'Ymin'], skiprows=1)

shapes = pd.read\_csv(os.path.join(data\_path, '3\_shapes.csv'))

def cache\_train\_b\_s():

    image\_set = extra\_functions.get\_class\_image(classes=[1, 2])

    num\_train = len(image\_set)

    print('num\_train\_images =', num\_train)

    train\_shapes = shapes[shapes['image\_id'].isin(image\_set)]

    image\_rows = train\_shapes['height'].min()

    image\_cols = train\_shapes['width'].min()

    num\_channels = 22

    num\_mask\_channels = 2

    f = h5py.File(os.path.join(data\_path, 'train\_b\_s.h5'), 'w')

    imgs = f.create\_dataset('train', (num\_train, num\_channels, image\_rows, image\_cols), dtype=np.float32, compression='gzip', compression\_opts=9)

    imgs\_mask = f.create\_dataset('train\_mask', (num\_train, num\_mask\_channels, image\_rows, image\_cols), dtype=np.uint8, compression='gzip', compression\_opts=9)

    ids = []

    i = 0

    for image\_id in image\_set:

        print(image\_id)

        image = extra\_functions.read\_image\_22(image\_id)

        height, width, \_ = image.shape

        imgs[i] = np.transpose(cv2.resize(image, (image\_cols, image\_rows), interpolation=cv2.INTER\_CUBIC), (2, 0, 1))

        imgs\_mask[i] = np.transpose(

            cv2.resize(np.transpose(extra\_functions.generate\_mask(image\_id, height, width, start=0,

                                                                  num\_mask\_channels=num\_mask\_channels,

                                                                  train=train\_wkt), (1, 2, 0)),

                       (image\_cols, image\_rows), interpolation=cv2.INTER\_CUBIC), (2, 0, 1))

        ids += [image\_id]

        i += 1

    # fix from there: https://github.com/h5py/h5py/issues/441

    f['train\_ids'] = np.array(ids).astype('|S9')

    f.close()

if \_\_name\_\_ == '\_\_main\_\_':

    cache\_train\_b\_s()

"""

Script that caches train data for future training

"""

from \_\_future\_\_ import division

import os

import pandas as pd

import extra\_functions

import h5py

import numpy as np

import cv2

data\_path = os.getcwd()

train\_wkt = pd.read\_csv(os.path.join(data\_path, 'train\_wkt\_v4.csv'))

gs = pd.read\_csv(os.path.join(data\_path, 'grid\_sizes.csv'), names=['ImageId', 'Xmax', 'Ymin'], skiprows=1)

shapes = pd.read\_csv(os.path.join(data\_path, '3\_shapes.csv'))

def cache\_train\_r\_t():

    image\_set = extra\_functions.get\_class\_image(classes=[3, 4])

    num\_train = len(image\_set)

    print('num\_train\_images =', num\_train)

    train\_shapes = shapes[shapes['image\_id'].isin(image\_set)]

    image\_rows = train\_shapes['height'].min()

    image\_cols = train\_shapes['width'].min()

    num\_channels = 22

    num\_mask\_channels = 2

    f = h5py.File(os.path.join(data\_path, 'train\_r\_t.h5'), 'w')

    imgs = f.create\_dataset('train', (num\_train, num\_channels, image\_rows, image\_cols), dtype=np.float32, compression='gzip', compression\_opts=9)

    imgs\_mask = f.create\_dataset('train\_mask', (num\_train, num\_mask\_channels, image\_rows, image\_cols), dtype=np.uint8, compression='gzip', compression\_opts=9)

    ids = []

    i = 0

    for image\_id in image\_set:

        print(image\_id)

        image = extra\_functions.read\_image\_22(image\_id)

        height, width, \_ = image.shape

        imgs[i] = np.transpose(cv2.resize(image, (image\_cols, image\_rows), interpolation=cv2.INTER\_CUBIC), (2, 0, 1))

        imgs\_mask[i] = np.transpose(

            cv2.resize(np.transpose(extra\_functions.generate\_mask(image\_id, height, width, start=2,

                                                                  num\_mask\_channels=num\_mask\_channels,

                                                                  train=train\_wkt), (1, 2, 0)),

                       (image\_cols, image\_rows), interpolation=cv2.INTER\_CUBIC), (2, 0, 1))

        ids += [image\_id]

        i += 1

    # fix from there: https://github.com/h5py/h5py/issues/441

    f['train\_ids'] = np.array(ids).astype('|S9')

    f.close()

if \_\_name\_\_ == '\_\_main\_\_':

    cache\_train\_r\_t()

"""

Script that caches train data for future training

"""

from \_\_future\_\_ import division

import os

import pandas as pd

import extra\_functions

import h5py

import numpy as np

import cv2

data\_path = os.getcwd()

train\_wkt = pd.read\_csv(os.path.join(data\_path, 'train\_wkt\_v4.csv'))

gs = pd.read\_csv(os.path.join(data\_path, 'grid\_sizes.csv'), names=['ImageId', 'Xmax', 'Ymin'], skiprows=1)

shapes = pd.read\_csv(os.path.join(data\_path, '3\_shapes.csv'))

def cache\_train\_t\_c():

    image\_set = extra\_functions.get\_class\_image(classes=[5, 6])

    num\_train = len(image\_set)

    print('num\_train\_images =', num\_train)

    train\_shapes = shapes[shapes['image\_id'].isin(image\_set)]

    image\_rows = train\_shapes['height'].min()

    image\_cols = train\_shapes['width'].min()

    num\_channels = 22

    num\_mask\_channels = 2

    f = h5py.File(os.path.join(data\_path, 'train\_t\_c.h5'), 'w')

    imgs = f.create\_dataset('train', (num\_train, num\_channels, image\_rows, image\_cols), dtype=np.float32, compression='gzip', compression\_opts=9)

    imgs\_mask = f.create\_dataset('train\_mask', (num\_train, num\_mask\_channels, image\_rows, image\_cols), dtype=np.uint8, compression='gzip', compression\_opts=9)

    ids = []

    i = 0

    for image\_id in image\_set:

        image = extra\_functions.read\_image\_22(image\_id)

        height, width, \_ = image.shape

        imgs[i] = np.transpose(cv2.resize(image, (image\_cols, image\_rows), interpolation=cv2.INTER\_CUBIC), (2, 0, 1))

        imgs\_mask[i] = np.transpose(

            cv2.resize(np.transpose(extra\_functions.generate\_mask(image\_id, height, width, start=4,

                                                                  num\_mask\_channels=num\_mask\_channels,

                                                                  train=train\_wkt), (1, 2, 0)),

                       (image\_cols, image\_rows), interpolation=cv2.INTER\_CUBIC), (2, 0, 1))

        ids += [image\_id]

        i += 1

    # fix from there: https://github.com/h5py/h5py/issues/441

    f['train\_ids'] = np.array(ids).astype('|S9')

    f.close()

if \_\_name\_\_ == '\_\_main\_\_':

    cache\_train\_t\_c()

"""

Script that caches train data for future training

"""

from \_\_future\_\_ import division

import os

import pandas as pd

import extra\_functions

import h5py

import numpy as np

import cv2

data\_path = os.getcwd()

train\_wkt = pd.read\_csv(os.path.join(data\_path, 'train\_wkt\_v4.csv'))

gs = pd.read\_csv(os.path.join(data\_path, 'grid\_sizes.csv'), names=['ImageId', 'Xmax', 'Ymin'], skiprows=1)

shapes = pd.read\_csv(os.path.join(data\_path, '3\_shapes.csv'))

def cache\_train\_vehicle():

    image\_set = extra\_functions.get\_class\_image(classes=[9, 10])

    num\_train = len(image\_set)

    print('num\_train\_images =', num\_train)

    train\_shapes = shapes[shapes['image\_id'].isin(image\_set)]

    image\_rows = train\_shapes['height'].min()

    image\_cols = train\_shapes['width'].min()

    num\_channels = 22

    num\_mask\_channels = 2

    f = h5py.File(os.path.join(data\_path, 'train\_vehicle.h5'), 'w')

    imgs = f.create\_dataset('train', (num\_train, num\_channels, image\_rows, image\_cols), dtype=np.float32, compression='gzip', compression\_opts=9)

    imgs\_mask = f.create\_dataset('train\_mask', (num\_train, num\_mask\_channels, image\_rows, image\_cols), dtype=np.uint8, compression='gzip', compression\_opts=9)

    ids = []

    i = 0

    for image\_id in image\_set:

        print(image\_id)

        image = extra\_functions.read\_image\_22(image\_id)

        height, width, \_ = image.shape

        imgs[i] = np.transpose(cv2.resize(image, (image\_cols, image\_rows), interpolation=cv2.INTER\_CUBIC), (2, 0, 1))

        imgs\_mask[i] = np.transpose(

            cv2.resize(np.transpose(extra\_functions.generate\_mask(image\_id, height, width, start=0,

                                                                  num\_mask\_channels=num\_mask\_channels,

                                                                  train=train\_wkt), (1, 2, 0)),

                       (image\_cols, image\_rows), interpolation=cv2.INTER\_CUBIC), (2, 0, 1))

        ids += [image\_id]

        i += 1

    # fix from there: https://github.com/h5py/h5py/issues/441

    f['train\_ids'] = np.array(ids).astype('|S9')

    f.close()

if \_\_name\_\_ == '\_\_main\_\_':

    cache\_train\_vehicle()

"""

Script that caches train data for future training

"""

from \_\_future\_\_ import division

import os

import pandas as pd

import extra\_functions

import h5py

import numpy as np

import cv2

data\_path = os.getcwd()

train\_wkt = pd.read\_csv(os.path.join(data\_path, 'train\_wkt\_v4.csv'))

gs = pd.read\_csv(os.path.join(data\_path, 'grid\_sizes.csv'), names=['ImageId', 'Xmax', 'Ymin'], skiprows=1)

shapes = pd.read\_csv(os.path.join(data\_path, '3\_shapes.csv'))

def cache\_train\_water():

    image\_set = extra\_functions.get\_class\_image(classes=[7, 8])

    num\_train = len(image\_set)

    print('num\_train\_images =', num\_train)

    train\_shapes = shapes[shapes['image\_id'].isin(image\_set)]

    image\_rows = train\_shapes['height'].min()

    image\_cols = train\_shapes['width'].min()

    num\_channels = 22

    num\_mask\_channels = 2

    f = h5py.File(os.path.join(data\_path, 'train\_water.h5'), 'w')

    imgs = f.create\_dataset('train', (num\_train, num\_channels, image\_rows, image\_cols), dtype=np.float32, compression='gzip', compression\_opts=9)

    imgs\_mask = f.create\_dataset('train\_mask', (num\_train, num\_mask\_channels, image\_rows, image\_cols), dtype=np.uint8, compression='gzip', compression\_opts=9)

    ids = []

    i = 0

    for image\_id in image\_set:

        image = extra\_functions.read\_image\_22(image\_id)

        height, width, \_ = image.shape

        imgs[i] = np.transpose(cv2.resize(image, (image\_cols, image\_rows), interpolation=cv2.INTER\_CUBIC), (2, 0, 1))

        imgs\_mask[i] = np.transpose(

            cv2.resize(np.transpose(extra\_functions.generate\_mask(image\_id, height, width, start=6,

                                                                  num\_mask\_channels=num\_mask\_channels,

                                                                  train=train\_wkt), (1, 2, 0)),

                       (image\_cols, image\_rows), interpolation=cv2.INTER\_CUBIC), (2, 0, 1))

        ids += [image\_id]

        i += 1

    # fix from there: https://github.com/h5py/h5py/issues/441

    f['train\_ids'] = np.array(ids).astype('|S9')

    f.close()

if \_\_name\_\_ == '\_\_main\_\_':

    cache\_train\_water()

## Modeling and Visualization

### U-Nets

from \_\_future\_\_ import division

import numpy as np

import keras

from keras.utils import Sequence

from keras.layers import concatenate, Conv2D, MaxPooling2D, UpSampling2D, Cropping2D, BatchNormalization

from keras import backend as K

import h5py

from keras.optimizers import Nadam

from keras.callbacks import ModelCheckpoint

from keras.backend import binary\_crossentropy

import datetime

import os

import random

import matplotlib.pyplot as plt

img\_rows = 112

img\_cols = 112

smooth = 1e-12

num\_channels = 22

num\_mask\_channels = 2

def jaccard\_coef(y\_true, y\_pred):

    intersection = K.sum(y\_true \* y\_pred, axis=[0, 1, 2])

    sum\_ = K.sum(y\_true + y\_pred, axis=[0, 1, 2])

    jac = (intersection + smooth) / (sum\_ - intersection + smooth)

    return K.mean(jac)

def jaccard\_coef\_int(y\_true, y\_pred):

    y\_pred\_pos = K.round(K.clip(y\_pred, 0, 1))

    intersection = K.sum(y\_true \* y\_pred\_pos, axis=[0, 1, 2])

    sum\_ = K.sum(y\_true + y\_pred\_pos, axis=[0, 1, 2])

    jac = (intersection + smooth) / (sum\_ - intersection + smooth)

    return K.mean(jac)

def jaccard\_coef\_loss(y\_true, y\_pred):

    return -K.log(jaccard\_coef(y\_true, y\_pred)) + binary\_crossentropy(y\_pred, y\_true)

def get\_unet0():

    inputs = keras.Input((img\_rows, img\_cols, num\_channels))

    conv1 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(inputs)

    conv1 = BatchNormalization()(conv1)

    conv1 = keras.layers.advanced\_activations.ELU()(conv1)

    conv1 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv1)

    conv1 = BatchNormalization()(conv1)

    conv1 = keras.layers.advanced\_activations.ELU()(conv1)

    pool1 = MaxPooling2D(pool\_size=(2, 2))(conv1)

    conv2 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool1)

    conv2 = BatchNormalization()(conv2)

    conv2 = keras.layers.advanced\_activations.ELU()(conv2)

    conv2 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv2)

    conv2 = BatchNormalization()(conv2)

    conv2 = keras.layers.advanced\_activations.ELU()(conv2)

    pool2 = MaxPooling2D(pool\_size=(2, 2))(conv2)

    conv3 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool2)

    conv3 = BatchNormalization()(conv3)

    conv3 = keras.layers.advanced\_activations.ELU()(conv3)

    conv3 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv3)

    conv3 = BatchNormalization()(conv3)

    conv3 = keras.layers.advanced\_activations.ELU()(conv3)

    pool3 = MaxPooling2D(pool\_size=(2, 2))(conv3)

    conv4 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool3)

    conv4 = BatchNormalization()(conv4)

    conv4 = keras.layers.advanced\_activations.ELU()(conv4)

    conv4 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv4)

    conv4 = BatchNormalization()(conv4)

    conv4 = keras.layers.advanced\_activations.ELU()(conv4)

    pool4 = MaxPooling2D(pool\_size=(2, 2))(conv4)

    conv5 = Conv2D(512, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool4)

    conv5 = BatchNormalization()(conv5)

    conv5 = keras.layers.advanced\_activations.ELU()(conv5)

    conv5 = Conv2D(512, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv5)

    conv5 = BatchNormalization()(conv5)

    conv5 = keras.layers.advanced\_activations.ELU()(conv5)

    up6 = concatenate([UpSampling2D(size=(2, 2))(conv5), conv4], axis=3)

    conv6 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up6)

    conv6 = BatchNormalization()(conv6)

    conv6 = keras.layers.advanced\_activations.ELU()(conv6)

    conv6 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv6)

    conv6 = BatchNormalization()(conv6)

    conv6 = keras.layers.advanced\_activations.ELU()(conv6)

    up7 = concatenate([UpSampling2D(size=(2, 2))(conv6), conv3], axis=3)

    conv7 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up7)

    conv7 = BatchNormalization()(conv7)

    conv7 = keras.layers.advanced\_activations.ELU()(conv7)

    conv7 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv7)

    conv7 = BatchNormalization()(conv7)

    conv7 = keras.layers.advanced\_activations.ELU()(conv7)

    up8 = concatenate([UpSampling2D(size=(2, 2))(conv7), conv2], axis=3)

    conv8 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up8)

    conv8 = BatchNormalization()(conv8)

    conv8 = keras.layers.advanced\_activations.ELU()(conv8)

    conv8 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv8)

    conv8 = BatchNormalization()(conv8)

    conv8 = keras.layers.advanced\_activations.ELU()(conv8)

    up9 = concatenate([UpSampling2D(size=(2, 2))(conv8), conv1], axis=3)

    conv9 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up9)

    conv9 = BatchNormalization()(conv9)

    conv9 = keras.layers.advanced\_activations.ELU()(conv9)

    conv9 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv9)

    crop9 = Cropping2D(cropping=((16, 16), (16, 16)))(conv9)

    conv9 = BatchNormalization()(crop9)

    conv9 = keras.layers.advanced\_activations.ELU()(conv9)

    conv10 = Conv2D(num\_mask\_channels, (1, 1), activation='sigmoid')(conv9)

    model = keras.Model(input=inputs, output=conv10)

    return model

def form\_batch(X, y, batch\_size):

    X\_batch = np.zeros((batch\_size, num\_channels, img\_rows, img\_cols))

    y\_batch = np.zeros((batch\_size, num\_mask\_channels, img\_rows-32, img\_cols-32))

    X\_height = X.shape[2]

    X\_width = X.shape[3]

    for i in range(batch\_size):

        random\_width = random.randint(0, X\_width - img\_cols - 1)

        random\_height = random.randint(0, X\_height - img\_rows - 1)

        random\_image = random.randint(0, X.shape[0] - 1)

        X\_batch[i] = X[random\_image, :, random\_height: random\_height + img\_rows, random\_width: random\_width + img\_cols]

        yb = y[random\_image, :, random\_height: random\_height + img\_rows, random\_width: random\_width + img\_cols]

        y\_batch[i] = yb[:, 16:16 + img\_rows - 32, 16:16 + img\_cols - 32]

    return np.transpose(X\_batch, (0, 2, 3, 1)), np.transpose(y\_batch, (0, 2, 3, 1))

class data\_generator(Sequence):

    def \_\_init\_\_(self, x\_set, y\_set, batch\_size, horizontal\_flip, vertical\_flip, swap\_axis):

        self.swap\_axis = swap\_axis

        self.vertical\_flip = vertical\_flip

        self.horizontal\_flip = horizontal\_flip

        self.x, self.y = x\_set, y\_set

        self.batch\_size = batch\_size

    def \_\_len\_\_(self):

        return int(np.ceil(len(self.x) / float(self.batch\_size)))

    def \_\_getitem\_\_(self, idx):

        X\_batch, y\_batch = form\_batch(self.x, self.y, self.batch\_size)

        for i in range(X\_batch.shape[0]):

            xb = X\_batch[i]

            yb = y\_batch[i]

            if self.horizontal\_flip:

                if np.random.random() < 0.5:

                    xb = np.fliplr(xb)

                    yb = np.fliplr(yb)

            if self.vertical\_flip:

                if np.random.random() < 0.5:

                    xb = np.flipud(xb)

                    yb = np.flipud(yb)

            if self.swap\_axis:

                if np.random.random() < 0.5:

                    xb = np.rot90(xb)

                    yb = np.rot90(yb)

            X\_batch[i] = xb

            y\_batch[i] = yb

        yield X\_batch, y\_batch

if \_\_name\_\_ == '\_\_main\_\_':

    data\_path = os.getcwd()

    now = datetime.datetime.now()

    print('[{}] Creating and compiling model...'.format(str(datetime.datetime.now())))

    model = get\_unet0()

    print('[{}] Reading train...'.format(str(datetime.datetime.now())))

    f = h5py.File(os.path.join(data\_path, 'train\_b\_s.h5'), 'r')

    X\_train = f['train']

    y\_train = np.array(f['train\_mask'])

    print(y\_train.shape)

    train\_ids = np.array(f['train\_ids'])

    batch\_size = 128

    nb\_epoch = 15

    filepath = "b\_s.h5"

    model.compile(optimizer=Nadam(lr=1e-3), loss=jaccard\_coef\_loss, metrics=['binary\_crossentropy', jaccard\_coef\_int])

    model.load\_weights('b\_s.h5')

    history = model.fit\_generator(generator=data\_generator(X\_train, y\_train, batch\_size, horizontal\_flip=True, vertical\_flip=True, swap\_axis=True),

                        epochs=nb\_epoch,

                        verbose=1,

                        samples\_per\_epoch=batch\_size \* 400,

                        validation\_data=data\_generator(X\_train, y\_train, 128, horizontal\_flip=False, vertical\_flip=False, swap\_axis=False),

                        validation\_steps = 4,

                        callbacks=[ModelCheckpoint(filepath, monitor="val\_loss", save\_best\_only=True, save\_weights\_only=True)],

                        workers=8

                        )

    # list all data in history

    print(history.history.keys())

    # summarize history for accuracy

    plt.plot(history.history['binary\_crossentropy'])

    plt.plot(history.history['val\_binary\_crossentropy'])

    plt.title('model binary\_crossentropy')

    plt.ylabel('binary\_crossentropy')

    plt.xlabel('epoch')

    plt.legend(['train', 'test'], loc='upper left')

    plt.savefig('b\_s\_binary\_crossentropy' +str(history.history['val\_jaccard\_coef\_int'][-1]) +'.png')

    # summarize history for loss

    plt.plot(history.history['loss'])

    plt.plot(history.history['val\_loss'])

    plt.title('model loss')

    plt.ylabel('loss')

    plt.xlabel('epoch')

    plt.legend(['train', 'test'], loc='upper left')

    plt.savefig('b\_s\_loss' +str(history.history['val\_jaccard\_coef\_int'][-1]) +'.png')

    f.close()

from \_\_future\_\_ import division

import numpy as np

import keras

from keras.utils import Sequence

from keras.layers import concatenate, Conv2D, MaxPooling2D, UpSampling2D, Cropping2D, BatchNormalization

from keras import backend as K

import h5py

from keras.optimizers import Nadam

from keras.callbacks import ModelCheckpoint

from keras.backend import binary\_crossentropy

import datetime

import os

import random

import matplotlib.pyplot as plt

img\_rows = 112

img\_cols = 112

smooth = 1e-12

num\_channels = 22

num\_mask\_channels = 2

def jaccard\_coef(y\_true, y\_pred):

    intersection = K.sum(y\_true \* y\_pred, axis=[0, 1, 2])

    sum\_ = K.sum(y\_true + y\_pred, axis=[0, 1, 2])

    jac = (intersection + smooth) / (sum\_ - intersection + smooth)

    return K.mean(jac)

def jaccard\_coef\_int(y\_true, y\_pred):

    y\_pred\_pos = K.round(K.clip(y\_pred, 0, 1))

    intersection = K.sum(y\_true \* y\_pred\_pos, axis=[0, 1, 2])

    sum\_ = K.sum(y\_true + y\_pred\_pos, axis=[0, 1, 2])

    jac = (intersection + smooth) / (sum\_ - intersection + smooth)

    return K.mean(jac)

def jaccard\_coef\_loss(y\_true, y\_pred):

    return -K.log(jaccard\_coef(y\_true, y\_pred)) + binary\_crossentropy(y\_pred, y\_true)

def get\_unet0():

    inputs = keras.Input((img\_rows, img\_cols, num\_channels))

    conv1 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(inputs)

    conv1 = BatchNormalization()(conv1)

    conv1 = keras.layers.advanced\_activations.ELU()(conv1)

    conv1 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv1)

    conv1 = BatchNormalization()(conv1)

    conv1 = keras.layers.advanced\_activations.ELU()(conv1)

    pool1 = MaxPooling2D(pool\_size=(2, 2))(conv1)

    conv2 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool1)

    conv2 = BatchNormalization()(conv2)

    conv2 = keras.layers.advanced\_activations.ELU()(conv2)

    conv2 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv2)

    conv2 = BatchNormalization()(conv2)

    conv2 = keras.layers.advanced\_activations.ELU()(conv2)

    pool2 = MaxPooling2D(pool\_size=(2, 2))(conv2)

    conv3 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool2)

    conv3 = BatchNormalization()(conv3)

    conv3 = keras.layers.advanced\_activations.ELU()(conv3)

    conv3 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv3)

    conv3 = BatchNormalization()(conv3)

    conv3 = keras.layers.advanced\_activations.ELU()(conv3)

    pool3 = MaxPooling2D(pool\_size=(2, 2))(conv3)

    conv4 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool3)

    conv4 = BatchNormalization()(conv4)

    conv4 = keras.layers.advanced\_activations.ELU()(conv4)

    conv4 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv4)

    conv4 = BatchNormalization()(conv4)

    conv4 = keras.layers.advanced\_activations.ELU()(conv4)

    pool4 = MaxPooling2D(pool\_size=(2, 2))(conv4)

    conv5 = Conv2D(512, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool4)

    conv5 = BatchNormalization()(conv5)

    conv5 = keras.layers.advanced\_activations.ELU()(conv5)

    conv5 = Conv2D(512, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv5)

    conv5 = BatchNormalization()(conv5)

    conv5 = keras.layers.advanced\_activations.ELU()(conv5)

    up6 = concatenate([UpSampling2D(size=(2, 2))(conv5), conv4], axis=3)

    conv6 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up6)

    conv6 = BatchNormalization()(conv6)

    conv6 = keras.layers.advanced\_activations.ELU()(conv6)

    conv6 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv6)

    conv6 = BatchNormalization()(conv6)

    conv6 = keras.layers.advanced\_activations.ELU()(conv6)

    up7 = concatenate([UpSampling2D(size=(2, 2))(conv6), conv3], axis=3)

    conv7 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up7)

    conv7 = BatchNormalization()(conv7)

    conv7 = keras.layers.advanced\_activations.ELU()(conv7)

    conv7 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv7)

    conv7 = BatchNormalization()(conv7)

    conv7 = keras.layers.advanced\_activations.ELU()(conv7)

    up8 = concatenate([UpSampling2D(size=(2, 2))(conv7), conv2], axis=3)

    conv8 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up8)

    conv8 = BatchNormalization()(conv8)

    conv8 = keras.layers.advanced\_activations.ELU()(conv8)

    conv8 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv8)

    conv8 = BatchNormalization()(conv8)

    conv8 = keras.layers.advanced\_activations.ELU()(conv8)

    up9 = concatenate([UpSampling2D(size=(2, 2))(conv8), conv1], axis=3)

    conv9 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up9)

    conv9 = BatchNormalization()(conv9)

    conv9 = keras.layers.advanced\_activations.ELU()(conv9)

    conv9 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv9)

    crop9 = Cropping2D(cropping=((16, 16), (16, 16)))(conv9)

    conv9 = BatchNormalization()(crop9)

    conv9 = keras.layers.advanced\_activations.ELU()(conv9)

    conv10 = Conv2D(num\_mask\_channels, (1, 1), activation='sigmoid')(conv9)

    model = keras.Model(input=inputs, output=conv10)

    return model

def form\_batch(X, y, batch\_size):

    X\_batch = np.zeros((batch\_size, num\_channels, img\_rows, img\_cols))

    y\_batch = np.zeros((batch\_size, num\_mask\_channels, img\_rows-32, img\_cols-32))

    X\_height = X.shape[2]

    X\_width = X.shape[3]

    for i in range(batch\_size):

        random\_width = random.randint(0, X\_width - img\_cols - 1)

        random\_height = random.randint(0, X\_height - img\_rows - 1)

        random\_image = random.randint(0, X.shape[0] - 1)

        X\_batch[i] = X[random\_image, :, random\_height: random\_height + img\_rows, random\_width: random\_width + img\_cols]

        yb = y[random\_image, :, random\_height: random\_height + img\_rows, random\_width: random\_width + img\_cols]

        y\_batch[i] = yb[:, 16:16 + img\_rows - 32, 16:16 + img\_cols - 32]

    return np.transpose(X\_batch, (0, 2, 3, 1)), np.transpose(y\_batch, (0, 2, 3, 1))

class data\_generator(Sequence):

    def \_\_init\_\_(self, x\_set, y\_set, batch\_size, horizontal\_flip, vertical\_flip, swap\_axis):

        self.swap\_axis = swap\_axis

        self.vertical\_flip = vertical\_flip

        self.horizontal\_flip = horizontal\_flip

        self.x, self.y = x\_set, y\_set

        self.batch\_size = batch\_size

    def \_\_len\_\_(self):

        return int(np.ceil(len(self.x) / float(self.batch\_size)))

    def \_\_getitem\_\_(self, idx):

        X\_batch, y\_batch = form\_batch(self.x, self.y, self.batch\_size)

        for i in range(X\_batch.shape[0]):

            xb = X\_batch[i]

            yb = y\_batch[i]

            if self.horizontal\_flip:

                if np.random.random() < 0.5:

                    xb = np.fliplr(xb)

                    yb = np.fliplr(yb)

            if self.vertical\_flip:

                if np.random.random() < 0.5:

                    xb = np.flipud(xb)

                    yb = np.flipud(yb)

            if self.swap\_axis:

                if np.random.random() < 0.5:

                    xb = np.rot90(xb)

                    yb = np.rot90(yb)

            X\_batch[i] = xb

            y\_batch[i] = yb

        return X\_batch, y\_batch #Changed this from yield to return for running the same file

if \_\_name\_\_ == '\_\_main\_\_':

    data\_path = os.getcwd()

    now = datetime.datetime.now()

    print('[{}] Creating and compiling model...'.format(str(datetime.datetime.now())))

    model = get\_unet0()

    print('[{}] Reading train...'.format(str(datetime.datetime.now())))

    f = h5py.File(os.path.join(data\_path, 'train\_water.h5'), 'r')

    X\_train = f['train']

    y\_train = np.array(f['train\_mask'])

    print(y\_train.shape)

    train\_ids = np.array(f['train\_ids'])

    batch\_size = 128

    nb\_epoch = 4

    filepath = "water.h5"

    model.compile(optimizer=Nadam(lr=1e-3), loss=jaccard\_coef\_loss, metrics=['binary\_crossentropy', jaccard\_coef\_int])

    #model.load\_weights('water.h5') #Comment this if you have not already run the model at least once, it helps to save time in subsequent training steps.

    history = model.fit\_generator(generator=data\_generator(X\_train, y\_train, batch\_size, horizontal\_flip=True, vertical\_flip=True, swap\_axis=True),

                    epochs=nb\_epoch,

                    verbose=1,

                    samples\_per\_epoch=batch\_size \* 100,

                    validation\_data=data\_generator(X\_train, y\_train, 128, horizontal\_flip=False, vertical\_flip=False, swap\_axis=False),

                    validation\_steps = 4,

                    callbacks=[ModelCheckpoint(filepath, monitor="val\_loss", save\_best\_only=True, save\_weights\_only=True)],

                    workers=8

                    )

    # list all data in history

    print(history.history.keys())

    # summarize history for accuracy

    plt.plot(history.history['binary\_crossentropy'])

    plt.plot(history.history['val\_binary\_crossentropy'])

    plt.title('model binary\_crossentropy')

    plt.ylabel('binary\_crossentropy')

    plt.xlabel('epoch')

    plt.legend(['train', 'test'], loc='upper left')

    plt.savefig('water\_binary\_crossentropy' +str(history.history['val\_jaccard\_coef\_int'][-1]) +'.png')

    # summarize history for loss

    plt.plot(history.history['loss'])

    plt.plot(history.history['val\_loss'])

    plt.title('model loss')

    plt.ylabel('loss')

    plt.xlabel('epoch')

    plt.legend(['train', 'test'], loc='upper left')

    plt.savefig('water\_loss' +str(history.history['val\_jaccard\_coef\_int'][-1]) +'.png')

    f.close()

print(history.history.keys())

from \_\_future\_\_ import division

import numpy as np

import keras

from keras.utils import Sequence

from keras.layers import concatenate, Conv2D, MaxPooling2D, UpSampling2D, Cropping2D, BatchNormalization

from keras import backend as K

import h5py

from keras.optimizers import Nadam

from keras.callbacks import ModelCheckpoint

from keras.backend import binary\_crossentropy

import datetime

import os

import random

import matplotlib.pyplot as plt

img\_rows = 112

img\_cols = 112

smooth = 1e-12

num\_channels = 22

num\_mask\_channels = 2

def jaccard\_coef(y\_true, y\_pred):

    intersection = K.sum(y\_true \* y\_pred, axis=[0, 1, 2])

    sum\_ = K.sum(y\_true + y\_pred, axis=[0, 1, 2])

    jac = (intersection + smooth) / (sum\_ - intersection + smooth)

    return K.mean(jac)

def jaccard\_coef\_int(y\_true, y\_pred):

    y\_pred\_pos = K.round(K.clip(y\_pred, 0, 1))

    intersection = K.sum(y\_true \* y\_pred\_pos, axis=[0, 1, 2])

    sum\_ = K.sum(y\_true + y\_pred\_pos, axis=[0, 1, 2])

    jac = (intersection + smooth) / (sum\_ - intersection + smooth)

    return K.mean(jac)

def jaccard\_coef\_loss(y\_true, y\_pred):

    return -K.log(jaccard\_coef(y\_true, y\_pred)) + binary\_crossentropy(y\_pred, y\_true)

def get\_unet0():

    inputs = keras.Input((img\_rows, img\_cols, num\_channels))

    conv1 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(inputs)

    conv1 = BatchNormalization()(conv1)

    conv1 = keras.layers.advanced\_activations.ELU()(conv1)

    conv1 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv1)

    conv1 = BatchNormalization()(conv1)

    conv1 = keras.layers.advanced\_activations.ELU()(conv1)

    pool1 = MaxPooling2D(pool\_size=(2, 2))(conv1)

    conv2 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool1)

    conv2 = BatchNormalization()(conv2)

    conv2 = keras.layers.advanced\_activations.ELU()(conv2)

    conv2 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv2)

    conv2 = BatchNormalization()(conv2)

    conv2 = keras.layers.advanced\_activations.ELU()(conv2)

    pool2 = MaxPooling2D(pool\_size=(2, 2))(conv2)

    conv3 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool2)

    conv3 = BatchNormalization()(conv3)

    conv3 = keras.layers.advanced\_activations.ELU()(conv3)

    conv3 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv3)

    conv3 = BatchNormalization()(conv3)

    conv3 = keras.layers.advanced\_activations.ELU()(conv3)

    pool3 = MaxPooling2D(pool\_size=(2, 2))(conv3)

    conv4 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool3)

    conv4 = BatchNormalization()(conv4)

    conv4 = keras.layers.advanced\_activations.ELU()(conv4)

    conv4 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv4)

    conv4 = BatchNormalization()(conv4)

    conv4 = keras.layers.advanced\_activations.ELU()(conv4)

    pool4 = MaxPooling2D(pool\_size=(2, 2))(conv4)

    conv5 = Conv2D(512, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool4)

    conv5 = BatchNormalization()(conv5)

    conv5 = keras.layers.advanced\_activations.ELU()(conv5)

    conv5 = Conv2D(512, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv5)

    conv5 = BatchNormalization()(conv5)

    conv5 = keras.layers.advanced\_activations.ELU()(conv5)

    up6 = concatenate([UpSampling2D(size=(2, 2))(conv5), conv4], axis=3)

    conv6 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up6)

    conv6 = BatchNormalization()(conv6)

    conv6 = keras.layers.advanced\_activations.ELU()(conv6)

    conv6 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv6)

    conv6 = BatchNormalization()(conv6)

    conv6 = keras.layers.advanced\_activations.ELU()(conv6)

    up7 = concatenate([UpSampling2D(size=(2, 2))(conv6), conv3], axis=3)

    conv7 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up7)

    conv7 = BatchNormalization()(conv7)

    conv7 = keras.layers.advanced\_activations.ELU()(conv7)

    conv7 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv7)

    conv7 = BatchNormalization()(conv7)

    conv7 = keras.layers.advanced\_activations.ELU()(conv7)

    up8 = concatenate([UpSampling2D(size=(2, 2))(conv7), conv2], axis=3)

    conv8 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up8)

    conv8 = BatchNormalization()(conv8)

    conv8 = keras.layers.advanced\_activations.ELU()(conv8)

    conv8 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv8)

    conv8 = BatchNormalization()(conv8)

    conv8 = keras.layers.advanced\_activations.ELU()(conv8)

    up9 = concatenate([UpSampling2D(size=(2, 2))(conv8), conv1], axis=3)

    conv9 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up9)

    conv9 = BatchNormalization()(conv9)

    conv9 = keras.layers.advanced\_activations.ELU()(conv9)

    conv9 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv9)

    crop9 = Cropping2D(cropping=((16, 16), (16, 16)))(conv9)

    conv9 = BatchNormalization()(crop9)

    conv9 = keras.layers.advanced\_activations.ELU()(conv9)

    conv10 = Conv2D(num\_mask\_channels, (1, 1), activation='sigmoid')(conv9)

    model = keras.Model(input=inputs, output=conv10)

    return model

def form\_batch(X, y, batch\_size):

    X\_batch = np.zeros((batch\_size, num\_channels, img\_rows, img\_cols))

    y\_batch = np.zeros((batch\_size, num\_mask\_channels, img\_rows-32, img\_cols-32))

    X\_height = X.shape[2]

    X\_width = X.shape[3]

    for i in range(batch\_size):

        random\_width = random.randint(0, X\_width - img\_cols - 1)

        random\_height = random.randint(0, X\_height - img\_rows - 1)

        random\_image = random.randint(0, X.shape[0] - 1)

        X\_batch[i] = X[random\_image, :, random\_height: random\_height + img\_rows, random\_width: random\_width + img\_cols]

        yb = y[random\_image, :, random\_height: random\_height + img\_rows, random\_width: random\_width + img\_cols]

        y\_batch[i] = yb[:, 16:16 + img\_rows - 32, 16:16 + img\_cols - 32]

    return np.transpose(X\_batch, (0, 2, 3, 1)), np.transpose(y\_batch, (0, 2, 3, 1))

class data\_generator(Sequence):

    def \_\_init\_\_(self, x\_set, y\_set, batch\_size, horizontal\_flip, vertical\_flip, swap\_axis):

        self.swap\_axis = swap\_axis

        self.vertical\_flip = vertical\_flip

        self.horizontal\_flip = horizontal\_flip

        self.x, self.y = x\_set, y\_set

        self.batch\_size = batch\_size

    def \_\_len\_\_(self):

        return int(np.ceil(len(self.x) / float(self.batch\_size)))

    def \_\_getitem\_\_(self, idx):

        X\_batch, y\_batch = form\_batch(self.x, self.y, self.batch\_size)

        for i in range(X\_batch.shape[0]):

            xb = X\_batch[i]

            yb = y\_batch[i]

            if self.horizontal\_flip:

                if np.random.random() < 0.5:

                    xb = np.fliplr(xb)

                    yb = np.fliplr(yb)

            if self.vertical\_flip:

                if np.random.random() < 0.5:

                    xb = np.flipud(xb)

                    yb = np.flipud(yb)

            if self.swap\_axis:

                if np.random.random() < 0.5:

                    xb = np.rot90(xb)

                    yb = np.rot90(yb)

            X\_batch[i] = xb

            y\_batch[i] = yb

        return X\_batch, y\_batch #Changed this from yield to return for running the same file

if \_\_name\_\_ == '\_\_main\_\_':

    data\_path = os.getcwd()

    now = datetime.datetime.now()

    print('[{}] Creating and compiling model...'.format(str(datetime.datetime.now())))

    model = get\_unet0()

    print('[{}] Reading train...'.format(str(datetime.datetime.now())))

    f = h5py.File(os.path.join(data\_path, 'train\_vehicle.h5'), 'r')

    X\_train = f['train']

    y\_train = np.array(f['train\_mask'])

    print(y\_train.shape)

    train\_ids = np.array(f['train\_ids'])

    batch\_size = 128

    nb\_epoch = 4

    inputs = data\_generator(X\_train, y\_train, batch\_size, horizontal\_flip=True, vertical\_flip=True, swap\_axis=True)

    print(inputs.x)

    filepath = "vehicle.h5"

    model.compile(optimizer=Nadam(lr=1e-3), loss=jaccard\_coef\_loss, metrics=['binary\_crossentropy', jaccard\_coef\_int])

    model.load\_weights('vehicle.h5') #Comment this if you have not already run the model at least once, it helps to save time in subsequent training steps.

    outputs = model.predict(X\_train)

    print(outputs)

    """

    history = model.fit\_generator(generator=data\_generator(X\_train, y\_train, batch\_size, horizontal\_flip=True, vertical\_flip=True, swap\_axis=True),

                    epochs=nb\_epoch,

                    verbose=1,

                    samples\_per\_epoch=batch\_size \* 200,

                    validation\_data=data\_generator(X\_train, y\_train, 128, horizontal\_flip=False, vertical\_flip=False, swap\_axis=False),

                    validation\_steps = 4,

                    callbacks=[ModelCheckpoint(filepath, monitor="val\_loss", save\_best\_only=True, save\_weights\_only=True)],

                    workers=8

                    )

    # list all data in history

    print(history.history.keys())

    # summarize history for accuracy

    plt.plot(history.history['binary\_crossentropy'])

    plt.plot(history.history['val\_binary\_crossentropy'])

    plt.title('model binary\_crossentropy')

    plt.ylabel('binary\_crossentropy')

    plt.xlabel('epoch')

    plt.legend(['train', 'test'], loc='upper left')

    plt.savefig('vehicle\_binary\_crossentropy' +str(history.history['val\_jaccard\_coef\_int'][-1]) +'.png')

    # summarize history for loss

    plt.plot(history.history['loss'])

    plt.plot(history.history['val\_loss'])

    plt.title('model loss')

    plt.ylabel('loss')

    plt.xlabel('epoch')

    plt.legend(['train', 'test'], loc='upper left')

    plt.savefig('vehicle\_loss' +str(history.history['val\_jaccard\_coef\_int'][-1]) +'.png')

    f.close()

"""

from \_\_future\_\_ import division

import numpy as np

import keras

from keras.utils import Sequence

from keras.layers import concatenate, Conv2D, MaxPooling2D, UpSampling2D, Cropping2D, BatchNormalization

from keras import backend as K

import h5py

from keras.optimizers import Nadam

from keras.callbacks import ModelCheckpoint

from keras.backend import binary\_crossentropy

import datetime

import os

import random

import matplotlib.pyplot as plt

img\_rows = 112

img\_cols = 112

smooth = 1e-12

num\_channels = 22

num\_mask\_channels = 2

def jaccard\_coef(y\_true, y\_pred):

    intersection = K.sum(y\_true \* y\_pred, axis=[0, 1, 2])

    sum\_ = K.sum(y\_true + y\_pred, axis=[0, 1, 2])

    jac = (intersection + smooth) / (sum\_ - intersection + smooth)

    return K.mean(jac)

def jaccard\_coef\_int(y\_true, y\_pred):

    y\_pred\_pos = K.round(K.clip(y\_pred, 0, 1))

    intersection = K.sum(y\_true \* y\_pred\_pos, axis=[0, 1, 2])

    sum\_ = K.sum(y\_true + y\_pred\_pos, axis=[0, 1, 2])

    jac = (intersection + smooth) / (sum\_ - intersection + smooth)

    return K.mean(jac)

def jaccard\_coef\_loss(y\_true, y\_pred):

    return -K.log(jaccard\_coef(y\_true, y\_pred)) + binary\_crossentropy(y\_pred, y\_true)

def get\_unet0():

    inputs = keras.Input((img\_rows, img\_cols, num\_channels))

    conv1 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(inputs)

    conv1 = BatchNormalization()(conv1)

    conv1 = keras.layers.advanced\_activations.ELU()(conv1)

    conv1 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv1)

    conv1 = BatchNormalization()(conv1)

    conv1 = keras.layers.advanced\_activations.ELU()(conv1)

    pool1 = MaxPooling2D(pool\_size=(2, 2))(conv1)

    conv2 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool1)

    conv2 = BatchNormalization()(conv2)

    conv2 = keras.layers.advanced\_activations.ELU()(conv2)

    conv2 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv2)

    conv2 = BatchNormalization()(conv2)

    conv2 = keras.layers.advanced\_activations.ELU()(conv2)

    pool2 = MaxPooling2D(pool\_size=(2, 2))(conv2)

    conv3 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool2)

    conv3 = BatchNormalization()(conv3)

    conv3 = keras.layers.advanced\_activations.ELU()(conv3)

    conv3 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv3)

    conv3 = BatchNormalization()(conv3)

    conv3 = keras.layers.advanced\_activations.ELU()(conv3)

    pool3 = MaxPooling2D(pool\_size=(2, 2))(conv3)

    conv4 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool3)

    conv4 = BatchNormalization()(conv4)

    conv4 = keras.layers.advanced\_activations.ELU()(conv4)

    conv4 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv4)

    conv4 = BatchNormalization()(conv4)

    conv4 = keras.layers.advanced\_activations.ELU()(conv4)

    pool4 = MaxPooling2D(pool\_size=(2, 2))(conv4)

    conv5 = Conv2D(512, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool4)

    conv5 = BatchNormalization()(conv5)

    conv5 = keras.layers.advanced\_activations.ELU()(conv5)

    conv5 = Conv2D(512, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv5)

    conv5 = BatchNormalization()(conv5)

    conv5 = keras.layers.advanced\_activations.ELU()(conv5)

    up6 = concatenate([UpSampling2D(size=(2, 2))(conv5), conv4], axis=3)

    conv6 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up6)

    conv6 = BatchNormalization()(conv6)

    conv6 = keras.layers.advanced\_activations.ELU()(conv6)

    conv6 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv6)

    conv6 = BatchNormalization()(conv6)

    conv6 = keras.layers.advanced\_activations.ELU()(conv6)

    up7 = concatenate([UpSampling2D(size=(2, 2))(conv6), conv3], axis=3)

    conv7 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up7)

    conv7 = BatchNormalization()(conv7)

    conv7 = keras.layers.advanced\_activations.ELU()(conv7)

    conv7 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv7)

    conv7 = BatchNormalization()(conv7)

    conv7 = keras.layers.advanced\_activations.ELU()(conv7)

    up8 = concatenate([UpSampling2D(size=(2, 2))(conv7), conv2], axis=3)

    conv8 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up8)

    conv8 = BatchNormalization()(conv8)

    conv8 = keras.layers.advanced\_activations.ELU()(conv8)

    conv8 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv8)

    conv8 = BatchNormalization()(conv8)

    conv8 = keras.layers.advanced\_activations.ELU()(conv8)

    up9 = concatenate([UpSampling2D(size=(2, 2))(conv8), conv1], axis=3)

    conv9 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up9)

    conv9 = BatchNormalization()(conv9)

    conv9 = keras.layers.advanced\_activations.ELU()(conv9)

    conv9 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv9)

    crop9 = Cropping2D(cropping=((16, 16), (16, 16)))(conv9)

    conv9 = BatchNormalization()(crop9)

    conv9 = keras.layers.advanced\_activations.ELU()(conv9)

    conv10 = Conv2D(num\_mask\_channels, (1, 1), activation='sigmoid')(conv9)

    model = keras.Model(input=inputs, output=conv10)

    return model

def form\_batch(X, y, batch\_size):

    X\_batch = np.zeros((batch\_size, num\_channels, img\_rows, img\_cols))

    y\_batch = np.zeros((batch\_size, num\_mask\_channels, img\_rows-32, img\_cols-32))

    X\_height = X.shape[2]

    X\_width = X.shape[3]

    for i in range(batch\_size):

        random\_width = random.randint(0, X\_width - img\_cols - 1)

        random\_height = random.randint(0, X\_height - img\_rows - 1)

        random\_image = random.randint(0, X.shape[0] - 1)

        X\_batch[i] = X[random\_image, :, random\_height: random\_height + img\_rows, random\_width: random\_width + img\_cols]

        yb = y[random\_image, :, random\_height: random\_height + img\_rows, random\_width: random\_width + img\_cols]

        y\_batch[i] = yb[:, 16:16 + img\_rows - 32, 16:16 + img\_cols - 32]

    return np.transpose(X\_batch, (0, 2, 3, 1)), np.transpose(y\_batch, (0, 2, 3, 1))

class data\_generator(Sequence):

    def \_\_init\_\_(self, x\_set, y\_set, batch\_size, horizontal\_flip, vertical\_flip, swap\_axis):

        self.swap\_axis = swap\_axis

        self.vertical\_flip = vertical\_flip

        self.horizontal\_flip = horizontal\_flip

        self.x, self.y = x\_set, y\_set

        self.batch\_size = batch\_size

    def \_\_len\_\_(self):

        return int(np.ceil(len(self.x) / float(self.batch\_size)))

    def \_\_getitem\_\_(self, idx):

        X\_batch, y\_batch = form\_batch(self.x, self.y, self.batch\_size)

        for i in range(X\_batch.shape[0]):

            xb = X\_batch[i]

            yb = y\_batch[i]

            if self.horizontal\_flip:

                if np.random.random() < 0.5:

                    xb = np.fliplr(xb)

                    yb = np.fliplr(yb)

            if self.vertical\_flip:

                if np.random.random() < 0.5:

                    xb = np.flipud(xb)

                    yb = np.flipud(yb)

            if self.swap\_axis:

                if np.random.random() < 0.5:

                    xb = np.rot90(xb)

                    yb = np.rot90(yb)

            X\_batch[i] = xb

            y\_batch[i] = yb

        return X\_batch, y\_batch #Changed this from yield to return for running the same file

if \_\_name\_\_ == '\_\_main\_\_':

    data\_path = os.getcwd()

    now = datetime.datetime.now()

    print('[{}] Creating and compiling model...'.format(str(datetime.datetime.now())))

    model = get\_unet0()

    print('[{}] Reading train...'.format(str(datetime.datetime.now())))

    f = h5py.File(os.path.join(data\_path, 'train\_t\_c.h5'), 'r')

    X\_train = f['train']

    y\_train = np.array(f['train\_mask'])

    print(y\_train.shape)

    train\_ids = np.array(f['train\_ids'])

    batch\_size = 128

    nb\_epoch = 4

    filepath = "t\_c.h5"

    model.compile(optimizer=Nadam(lr=1e-3), loss=jaccard\_coef\_loss, metrics=['binary\_crossentropy', jaccard\_coef\_int])

    #model.load\_weights('t\_c.h5') #Comment this if you have not already run the model at least once, it helps to save time in subsequent training steps.

    history = model.fit\_generator(generator=data\_generator(X\_train, y\_train, batch\_size, horizontal\_flip=True, vertical\_flip=True, swap\_axis=True),

                    epochs=nb\_epoch,

                    verbose=1,

                    samples\_per\_epoch=batch\_size \* 100,

                    validation\_data=data\_generator(X\_train, y\_train, 128, horizontal\_flip=False, vertical\_flip=False, swap\_axis=False),

                    validation\_steps = 4,

                    callbacks=[ModelCheckpoint(filepath, monitor="val\_loss", save\_best\_only=True, save\_weights\_only=True)],

                    workers=8

                    )

    # list all data in history

    print(history.history.keys())

    # summarize history for accuracy

    plt.plot(history.history['binary\_crossentropy'])

    plt.plot(history.history['val\_binary\_crossentropy'])

    plt.title('model binary\_crossentropy')

    plt.ylabel('binary\_crossentropy')

    plt.xlabel('epoch')

    plt.legend(['train', 'test'], loc='upper left')

    plt.savefig('t\_c\_binary\_crossentropy' +str(history.history['val\_jaccard\_coef\_int'][-1]) +'.png')

    # summarize history for loss

    plt.plot(history.history['loss'])

    plt.plot(history.history['val\_loss'])

    plt.title('model loss')

    plt.ylabel('loss')

    plt.xlabel('epoch')

    plt.legend(['train', 'test'], loc='upper left')

    plt.savefig('t\_c\_loss' +str(history.history['val\_jaccard\_coef\_int'][-1]) +'.png')

    f.close()

### Predict and Visualize Files

import os

import h5py

import cv2

import numpy as np

import tifffile as tiff

data\_path = os.getcwd()

def stretch\_8bit(bands, lower\_percent=5, higher\_percent=95):

    out = np.zeros\_like(bands).astype(np.float32)

    for i in range(3):

        a = 0

        b = 1

        c = np.percentile(bands[:,:, i], lower\_percent)

        d = np.percentile(bands[:,:, i], higher\_percent)

        t = a + (bands[:,:, i] - c) \* (b - a) / (d - c)

        t[t<a] = a

        t[t>b] = b

        out[:,:, i] =t

    return out.astype(np.float32)

#rgb = tiff.imread(data\_path + '/three\_band/6110\_4\_0.tif')

#rgb = np.rollaxis(rgb, 0, 3)

#cv2.imwrite('org.png',255 \* stretch\_8bit(rgb))

f = h5py.File(os.path.join(data\_path, 'train\_test.h5'), 'r')

X\_train = f['train\_mask'][0]

#print(f['train\_ids'])

img = np.transpose(X\_train, (1, 2, 0))

#img = img[:,:, 21:]

img = np.concatenate([img, np.expand\_dims(img[:, :, 0], 2)], axis=2)

img = 255 \* img

img = img.astype(np.uint8)

cv2.imwrite('mask.png',img)

"""

Code to visualize individual images, listed as real\_test\_ids

"""

import os

import h5py

import cv2

import numpy as np

import tifffile as tiff

data\_path = os.getcwd()

def stretch\_8bit(bands, lower\_percent=5, higher\_percent=95):

    out = np.zeros\_like(bands).astype(np.float32)

    for i in range(3):

        a = 0

        b = 1

        c = np.percentile(bands[:,:, i], lower\_percent)

        d = np.percentile(bands[:,:, i], higher\_percent)

        t = a + (bands[:,:, i] - c) \* (b - a) / (d - c)

        t[t<a] = a

        t[t>b] = b

        out[:,:, i] =t

    return out.astype(np.float32)

real\_test\_ids = ['6080\_4\_4', '6080\_4\_1', '6010\_0\_1', '6150\_3\_4', '6020\_0\_4', '6020\_4\_3',

                 '6150\_4\_3', '6070\_3\_4', '6020\_1\_3', '6060\_1\_4', '6050\_4\_4', '6110\_2\_3',

                 '6060\_4\_1', '6100\_2\_4', '6050\_3\_3', '6100\_0\_2', '6060\_0\_0', '6060\_0\_1',

                 '6060\_0\_3', '6060\_2\_0', '6120\_1\_4', '6160\_1\_4', '6120\_3\_3', '6140\_2\_3',

                 '6090\_3\_2', '6090\_3\_4', '6170\_4\_4', '6120\_4\_4', '6030\_1\_4', '6120\_0\_2',

                 '6030\_1\_2', '6160\_0\_0']

for i in real\_test\_ids:

    rgb = tiff.imread(data\_path + '/three\_band/' + i +'.tif')

    rgb = np.rollaxis(rgb, 0, 3)

    cv2.imwrite(i+'.png', 255 \* stretch\_8bit(rgb))

#f = h5py.File(os.path.join(data\_path, 'train\_test.h5'), 'r')

#X\_train = f['train'][0]

#print(f['train\_ids'])

#img = np.transpose(X\_train, (1, 2, 0))

#img = img[:,:, 19:]

#img = 255 \* img

#img = img.astype(np.float32)

#cv2.imwrite('rgb.png',img)

import os

import pandas as pd

import numpy as np

import cv2

import extra\_functions

data\_path = os.getcwd()

num\_channels = 22

num\_mask\_channels = 2

pred = pd.read\_csv('temp\_b\_s.csv')

shapes = pd.read\_csv(os.path.join(data\_path, '3\_shapes.csv'))

#test\_id = pred['ImageId']

test\_id = ['6050\_4\_4', '6060\_0\_1', '6060\_1\_4', '6100\_0\_2', '6100\_2\_4', '6110\_2\_3', '6120\_1\_4', '6120\_3\_3']

for image\_id in test\_id:

    print(image\_id)

    mask = extra\_functions.generate\_mask(image\_id, int(shapes[shapes['image\_id'] == image\_id]['height']),

                                         int(shapes[shapes['image\_id'] == image\_id]['width']), start=0,

                                         num\_mask\_channels=num\_mask\_channels, train=pred)

    mask = np.transpose(mask, (1, 2, 0))

    mask = extra\_functions.stretch\_n(mask)

    img = np.concatenate([mask, np.expand\_dims(mask[:, :, 0], 2)], axis=2)

    img = 255 \* img

    img = img.astype(np.uint8)

    cv2.imwrite('mask' + image\_id +'.png', img)

"""

code to visualize the .h5 files cached for training

"""

import os

import h5py

import cv2

import numpy as np

import tifffile as tiff

data\_path = os.getcwd()

def stretch\_8bit(bands, lower\_percent=5, higher\_percent=95):

    out = np.zeros\_like(bands).astype(np.float32)

    for i in range(3):

        a = 0

        b = 1

        c = np.percentile(bands[:,:, i], lower\_percent)

        d = np.percentile(bands[:,:, i], higher\_percent)

        t = a + (bands[:,:, i] - c) \* (b - a) / (d - c)

        t[t<a] = a

        t[t>b] = b

        out[:,:, i] =t

    return out.astype(np.float32)

#rgb = tiff.imread(data\_path + '/three\_band/6110\_4\_0.tif')

#rgb = np.rollaxis(rgb, 0, 3)

#cv2.imwrite('org.png',255 \* stretch\_8bit(rgb))

f = h5py.File(os.path.join(data\_path, 'train\_test.h5'), 'r')

X\_train = f['train'][0]

#print(f['train\_ids'])

img = np.transpose(X\_train, (1, 2, 0))

img = img[:,:, 19:]

img = 255 \* img

img = img.astype(np.float32)

cv2.imwrite('rgb.png',img)

from \_\_future\_\_ import division

import numpy as np

import tensorflow.keras

from tensorflow.keras.utils import Sequence

import threading

from tensorflow.keras.layers import Input, concatenate, Conv2D, MaxPooling2D, UpSampling2D, Cropping2D, BatchNormalization

from tensorflow.keras import backend as K

import h5py

from tensorflow.keras.optimizers import Nadam

from tensorflow.keras.callbacks import ModelCheckpoint

from tensorflow.keras.losses import binary\_crossentropy

import datetime

import os

import random

import matplotlib.pyplot as plt

img\_rows = 112

img\_cols = 112

smooth = 1e-12

num\_channels = 22

num\_mask\_channels = 2

def jaccard\_coef(y\_true, y\_pred):

    intersection = K.sum(y\_true \* y\_pred, axis=[0, 1, 2])

    sum\_ = K.sum(y\_true + y\_pred, axis=[0, 1, 2])

    jac = (intersection + smooth) / (sum\_ - intersection + smooth)

    return K.mean(jac)

def jaccard\_coef\_int(y\_true, y\_pred):

    y\_pred\_pos = K.round(K.clip(y\_pred, 0, 1))

    intersection = K.sum(y\_true \* y\_pred\_pos, axis=[0, 1, 2])

    sum\_ = K.sum(y\_true + y\_pred\_pos, axis=[0, 1, 2])

    jac = (intersection + smooth) / (sum\_ - intersection + smooth)

    return K.mean(jac)

def jaccard\_coef\_loss(y\_true, y\_pred):

    return -K.log(jaccard\_coef(y\_true, y\_pred)) + binary\_crossentropy(y\_pred, y\_true)

def get\_unet0():

    inputs = Input((img\_rows, img\_cols, num\_channels))

    conv1 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(inputs)

    conv1 = BatchNormalization()(conv1)

    conv1 = tensorflow.keras.layers.ELU()(conv1)

    conv1 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv1)

    conv1 = BatchNormalization()(conv1)

    conv1 = tensorflow.keras.layers.ELU()(conv1)

    pool1 = MaxPooling2D(pool\_size=(2, 2))(conv1)

    conv2 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool1)

    conv2 = BatchNormalization()(conv2)

    conv2 = tensorflow.keras.layers.ELU()(conv2)

    conv2 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv2)

    conv2 = BatchNormalization()(conv2)

    conv2 = tensorflow.keras.layers.ELU()(conv2)

    pool2 = MaxPooling2D(pool\_size=(2, 2))(conv2)

    conv3 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool2)

    conv3 = BatchNormalization()(conv3)

    conv3 = tensorflow.keras.layers.ELU()(conv3)

    conv3 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv3)

    conv3 = BatchNormalization()(conv3)

    conv3 = tensorflow.keras.layers.ELU()(conv3)

    pool3 = MaxPooling2D(pool\_size=(2, 2))(conv3)

    conv4 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool3)

    conv4 = BatchNormalization()(conv4)

    conv4 = tensorflow.keras.layers.ELU()(conv4)

    conv4 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv4)

    conv4 = BatchNormalization()(conv4)

    conv4 = tensorflow.keras.layers.ELU()(conv4)

    pool4 = MaxPooling2D(pool\_size=(2, 2))(conv4)

    conv5 = Conv2D(512, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool4)

    conv5 = BatchNormalization()(conv5)

    conv5 = tensorflow.keras.layers.ELU()(conv5)

    conv5 = Conv2D(512, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv5)

    conv5 = BatchNormalization()(conv5)

    conv5 = tensorflow.keras.layers.ELU()(conv5)

    up6 = concatenate([UpSampling2D(size=(2, 2))(conv5), conv4], axis=3)

    conv6 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up6)

    conv6 = BatchNormalization()(conv6)

    conv6 = tensorflow.keras.layers.ELU()(conv6)

    conv6 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv6)

    conv6 = BatchNormalization()(conv6)

    conv6 = tensorflow.keras.layers.ELU()(conv6)

    up7 = concatenate([UpSampling2D(size=(2, 2))(conv6), conv3], axis=3)

    conv7 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up7)

    conv7 = BatchNormalization()(conv7)

    conv7 = tensorflow.keras.layers.ELU()(conv7)

    conv7 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv7)

    conv7 = BatchNormalization()(conv7)

    conv7 = tensorflow.keras.layers.ELU()(conv7)

    up8 = concatenate([UpSampling2D(size=(2, 2))(conv7), conv2], axis=3)

    conv8 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up8)

    conv8 = BatchNormalization()(conv8)

    conv8 = tensorflow.keras.layers.ELU()(conv8)

    conv8 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv8)

    conv8 = BatchNormalization()(conv8)

    conv8 = tensorflow.keras.layers.ELU()(conv8)

    up9 = concatenate([UpSampling2D(size=(2, 2))(conv8), conv1], axis=3)

    conv9 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up9)

    conv9 = BatchNormalization()(conv9)

    conv9 = tensorflow.keras.layers.ELU()(conv9)

    conv9 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv9)

    crop9 = Cropping2D(cropping=((16, 16), (16, 16)))(conv9)

    conv9 = BatchNormalization()(crop9)

    conv9 = tensorflow.keras.layers.ELU()(conv9)

    conv10 = Conv2D(num\_mask\_channels, (1, 1), activation='sigmoid')(conv9)

    model = tensorflow.keras.Model(inputs=inputs, outputs=conv10)

    return model

def flip\_axis(x, axis):

    x = np.asarray(x).swapaxes(axis, 0)

    x = x[::-1, ...]

    x = x.swapaxes(0, axis)

    return x

def form\_batch(X, y, batch\_size):

    X\_batch = np.zeros((batch\_size, num\_channels, img\_rows, img\_cols))

    y\_batch = np.zeros((batch\_size, num\_mask\_channels, img\_rows-32, img\_cols-32))

    X\_height = X.shape[2]

    X\_width = X.shape[3]

    for i in range(batch\_size):

        random\_width = random.randint(0, X\_width - img\_cols - 1)

        random\_height = random.randint(0, X\_height - img\_rows - 1)

        random\_image = random.randint(0, X.shape[0] - 1)

        X\_batch[i] = X[random\_image, :, random\_height: random\_height + img\_rows, random\_width: random\_width + img\_cols]

        yb = y[random\_image, :, random\_height: random\_height + img\_rows, random\_width: random\_width + img\_cols]

        y\_batch[i] = yb[:, 16:16 + img\_rows - 32, 16:16 + img\_cols - 32]

    return np.transpose(X\_batch, (0, 2, 3, 1)), np.transpose(y\_batch, (0, 2, 3, 1))

class threadsafe\_iter:

    """Takes an iterator/generator and makes it thread-safe by

    serializing call to the `next` method of given iterator/generator.

    """

    def \_\_init\_\_(self, it):

        self.it = it

        self.lock = threading.Lock()

    def \_\_iter\_\_(self):

        return self

    def \_\_next\_\_(self): # Py3

        with self.lock:

            return next(self.it)

def threadsafe\_generator(f):

    """A decorator that takes a generator function and makes it thread-safe.

    """

    def g(\*a, \*\*kw):

        return threadsafe\_iter(f(\*a, \*\*kw))

    return g

@threadsafe\_generator

def mine\_hard\_samples(model, datagen, batch\_size):

    while True:

        samples, targets, loss = [], [], []

        x\_data, y\_data = next(datagen)

        preds = model.predict(x\_data)

        for i in range(len(preds)):

            loss.append(K.mean(jaccard\_coef\_loss(y\_data[i], preds[i])))

        ind = np.argpartition(np.asarray(loss), -int(batch\_size / 2))[-int(batch\_size / 2):]

        samples += x\_data[ind].tolist()

        targets += y\_data[ind].tolist()

        x\_data, y\_data = next(datagen)

        samples += x\_data[:int(batch\_size/2)].tolist()

        targets += y\_data[:int(batch\_size/2)].tolist()

        samples, targets = map(np.array, (samples, targets))

        for i in range(batch\_size):

            xb = samples[i]

            yb = targets[i]

            if np.random.random() < 0.5:

                xb = np.fliplr(xb)

                yb = np.fliplr(yb)

            if np.random.random() < 0.5:

                xb = np.flipud(xb)

                yb = np.flipud(yb)

            if np.random.random() < 0.5:

                xb = np.rot90(xb)

                yb = np.rot90(yb)

            samples[i] = xb

            targets[i] = yb

        yield samples, targets

@threadsafe\_generator

def gen(batch\_size):

    while True:

        x\_data, y\_data = form\_batch(X\_train, y\_train, batch\_size)

        yield x\_data, y\_data

if \_\_name\_\_ == '\_\_main\_\_':

    data\_path = os.getcwd()

    now = datetime.datetime.now()

    print('[{}] Creating and compiling model...'.format(str(datetime.datetime.now())))

    model = get\_unet0()

    print('[{}] Reading train...'.format(str(datetime.datetime.now())))

    f = h5py.File(os.path.join(data\_path, 'train\_b\_s.h5'), 'r')

    X\_train = f['train']

    y\_train = np.array(f['train\_mask'])

    print(y\_train.shape)

    train\_ids = np.array(f['train\_ids'])

    batch\_size = 128

    nb\_epoch = 5

    filepath = "b\_s.h5"

    model.compile(optimizer=Nadam(lr=1e-4), loss=jaccard\_coef\_loss, metrics=['binary\_crossentropy', jaccard\_coef\_int])

    model.load\_weights('b\_s.h5')

    x, y = next(gen(batch\_size))

    model.predict(x)

    history = model.fit\_generator(generator=mine\_hard\_samples(model, gen(batch\_size), batch\_size),

                                  epochs=nb\_epoch,

                                  verbose=1,

                                  steps\_per\_epoch=40,

                                  validation\_data=gen(batch\_size),

                                  validation\_steps=4,

                                  callbacks=[ModelCheckpoint(filepath, monitor="val\_loss", save\_best\_only=True, save\_weights\_only=True)],

                                  workers=8

                                  )

    plt.plot(history.history['binary\_crossentropy'])

    plt.plot(history.history['val\_binary\_crossentropy'])

    plt.title('model binary\_crossentropy')

    plt.ylabel('binary\_crossentropy')

    plt.xlabel('epoch')

    plt.legend(['train', 'test'], loc='upper left')

    plt.savefig('b\_s\_binary\_crossentropy' +str(np.min(history.history['val\_binary\_crossentropy'])) +'.png')

    # summarize history for loss

    plt.plot(history.history['loss'])

    plt.plot(history.history['val\_loss'])

    plt.title('model loss')

    plt.ylabel('loss')

    plt.xlabel('epoch')

    plt.legend(['train', 'test'], loc='upper left')

    plt.savefig('b\_s\_loss' +str(np.max(history.history['val\_jaccard\_coef\_int'])) +'.png')

    f.close()

"""

This code can be modified to visualize predicts of any model.

"""

from \_\_future\_\_ import division

import os

from tqdm import tqdm

import pandas as pd

import extra\_functions

import tensorflow.keras

from tensorflow.keras.layers import Input, concatenate, Conv2D, MaxPooling2D, UpSampling2D, Cropping2D, BatchNormalization

from tensorflow.keras.optimizers import Nadam

from tensorflow.keras import backend as K

from tensorflow.keras.optimizers import Nadam

from tensorflow.keras.losses import binary\_crossentropy

import shapely.geometry

from numba import jit

import numpy as np

img\_rows = 112

img\_cols = 112

smooth = 1e-12

data\_path = os.getcwd()

num\_channels = 22

num\_mask\_channels = 2

threashold = 0.3

def get\_unet0():

    inputs = Input((img\_rows, img\_cols, num\_channels))

    conv1 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(inputs)

    conv1 = BatchNormalization()(conv1)

    conv1 = tensorflow.keras.layers.ELU()(conv1)

    conv1 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv1)

    conv1 = BatchNormalization()(conv1)

    conv1 = tensorflow.keras.layers.ELU()(conv1)

    pool1 = MaxPooling2D(pool\_size=(2, 2))(conv1)

    conv2 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool1)

    conv2 = BatchNormalization()(conv2)

    conv2 = tensorflow.keras.layers.ELU()(conv2)

    conv2 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv2)

    conv2 = BatchNormalization()(conv2)

    conv2 = tensorflow.keras.layers.ELU()(conv2)

    pool2 = MaxPooling2D(pool\_size=(2, 2))(conv2)

    conv3 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool2)

    conv3 = BatchNormalization()(conv3)

    conv3 = tensorflow.keras.layers.ELU()(conv3)

    conv3 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv3)

    conv3 = BatchNormalization()(conv3)

    conv3 = tensorflow.keras.layers.ELU()(conv3)

    pool3 = MaxPooling2D(pool\_size=(2, 2))(conv3)

    conv4 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool3)

    conv4 = BatchNormalization()(conv4)

    conv4 = tensorflow.keras.layers.ELU()(conv4)

    conv4 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv4)

    conv4 = BatchNormalization()(conv4)

    conv4 = tensorflow.keras.layers.ELU()(conv4)

    pool4 = MaxPooling2D(pool\_size=(2, 2))(conv4)

    conv5 = Conv2D(512, (3, 3), padding='same', kernel\_initializer='he\_uniform')(pool4)

    conv5 = BatchNormalization()(conv5)

    conv5 = tensorflow.keras.layers.ELU()(conv5)

    conv5 = Conv2D(512, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv5)

    conv5 = BatchNormalization()(conv5)

    conv5 = tensorflow.keras.layers.ELU()(conv5)

    up6 = concatenate([UpSampling2D(size=(2, 2))(conv5), conv4], axis=3)

    conv6 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up6)

    conv6 = BatchNormalization()(conv6)

    conv6 = tensorflow.keras.layers.ELU()(conv6)

    conv6 = Conv2D(256, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv6)

    conv6 = BatchNormalization()(conv6)

    conv6 = tensorflow.keras.layers.ELU()(conv6)

    up7 = concatenate([UpSampling2D(size=(2, 2))(conv6), conv3], axis=3)

    conv7 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up7)

    conv7 = BatchNormalization()(conv7)

    conv7 = tensorflow.keras.layers.ELU()(conv7)

    conv7 = Conv2D(128, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv7)

    conv7 = BatchNormalization()(conv7)

    conv7 = tensorflow.keras.layers.ELU()(conv7)

    up8 = concatenate([UpSampling2D(size=(2, 2))(conv7), conv2], axis=3)

    conv8 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up8)

    conv8 = BatchNormalization()(conv8)

    conv8 = tensorflow.keras.layers.ELU()(conv8)

    conv8 = Conv2D(64, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv8)

    conv8 = BatchNormalization()(conv8)

    conv8 = tensorflow.keras.layers.ELU()(conv8)

    up9 = concatenate([UpSampling2D(size=(2, 2))(conv8), conv1], axis=3)

    conv9 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(up9)

    conv9 = BatchNormalization()(conv9)

    conv9 = tensorflow.keras.layers.ELU()(conv9)

    conv9 = Conv2D(32, (3, 3), padding='same', kernel\_initializer='he\_uniform')(conv9)

    crop9 = Cropping2D(cropping=((16, 16), (16, 16)))(conv9)

    conv9 = BatchNormalization()(crop9)

    conv9 = tensorflow.keras.layers.ELU()(conv9)

    conv10 = Conv2D(num\_mask\_channels, (1, 1), activation='sigmoid')(conv9)

    model = tensorflow.keras.Model(inputs=inputs, outputs=conv10)

    return model

def jaccard\_coef(y\_true, y\_pred):

    intersection = K.sum(y\_true \* y\_pred, axis=[0, 1, 2])

    sum\_ = K.sum(y\_true + y\_pred, axis=[0, 1, 2])

    jac = (intersection + smooth) / (sum\_ - intersection + smooth)

    return K.mean(jac)

def jaccard\_coef\_int(y\_true, y\_pred):

    y\_pred\_pos = K.round(K.clip(y\_pred, 0, 1))

    intersection = K.sum(y\_true \* y\_pred\_pos, axis=[0, 1, 2])

    sum\_ = K.sum(y\_true + y\_pred\_pos, axis=[0, 1, 2])

    jac = (intersection + smooth) / (sum\_ - intersection + smooth)

    return K.mean(jac)

def jaccard\_coef\_loss(y\_true, y\_pred):

    return -K.log(jaccard\_coef(y\_true, y\_pred)) + binary\_crossentropy(y\_pred, y\_true)

def read\_model(file):

    model = get\_unet0()

    model.compile(optimizer=Nadam(lr=1e-3), loss=jaccard\_coef\_loss, metrics=['binary\_crossentropy', jaccard\_coef\_int])

    model.load\_weights(file)

    return model

model = read\_model('b\_s.h5')

sample = pd.read\_csv('sample\_submission.csv')

three\_band\_path = os.path.join(data\_path, 'three\_band')

train\_wkt = pd.read\_csv(os.path.join(data\_path, 'train\_wkt\_v4.csv'))

gs = pd.read\_csv(os.path.join(data\_path, 'grid\_sizes.csv'), names=['ImageId', 'Xmax', 'Ymin'], skiprows=1)

shapes = pd.read\_csv(os.path.join(data\_path, '3\_shapes.csv'))

#test\_ids = shapes.loc[~shapes['image\_id'].isin(train\_wkt['ImageId'].unique()), 'image\_id']

test\_ids = ['6050\_4\_4', '6060\_0\_1', '6060\_1\_4', '6100\_0\_2', '6100\_2\_4', '6110\_2\_3', '6120\_1\_4', '6120\_3\_3']

result = []

@jit

def mask2poly(predicted\_mask, threashold, x\_scaler, y\_scaler):

    polygons = extra\_functions.mask2polygons\_layer(predicted\_mask > threashold, epsilon=0, min\_area=5)

    polygons = shapely.affinity.scale(polygons, xfact=1.0 / x\_scaler, yfact=1.0 / y\_scaler, origin=(0, 0, 0))

    return shapely.wkt.dumps(polygons.buffer(2.6e-5))

#for image\_id in tqdm(test\_ids[:2]):

for image\_id in test\_ids:

    print(image\_id)

    image = extra\_functions.read\_image\_22(image\_id)

    H = image.shape[0]

    W = image.shape[1]

    x\_max, y\_min = extra\_functions.\_get\_xmax\_ymin(image\_id)

    predicted\_mask = extra\_functions.make\_prediction\_cropped(model, image, initial\_size=(112, 112),

                                                             final\_size=(112-32, 112-32),

                                                             num\_masks=num\_mask\_channels, num\_channels=num\_channels)

    image\_v = np.flipud(image)

    predicted\_mask\_v = extra\_functions.make\_prediction\_cropped(model, image\_v, initial\_size=(112, 112),

                                                               final\_size=(112 - 32, 112 - 32),

                                                               num\_masks=2,

                                                               num\_channels=num\_channels)

    image\_h = np.fliplr(image)

    predicted\_mask\_h = extra\_functions.make\_prediction\_cropped(model, image\_h, initial\_size=(112, 112),

                                                               final\_size=(112 - 32, 112 - 32),

                                                               num\_masks=2,

                                                               num\_channels=num\_channels)

    image\_s = np.rot90(image)

    predicted\_mask\_s = extra\_functions.make\_prediction\_cropped(model, image\_s, initial\_size=(112, 112),

                                                               final\_size=(112 - 32, 112 - 32),

                                                               num\_masks=2,

                                                               num\_channels=num\_channels)

    new\_mask = np.power(predicted\_mask \*

                        np.flipud(predicted\_mask\_v) \*

                        np.fliplr(predicted\_mask\_h) \*

                        np.rot90(predicted\_mask\_s, 3), 0.25)

    x\_scaler, y\_scaler = extra\_functions.get\_scalers(H, W, x\_max, y\_min)

    mask\_channel = 0

    result += [(image\_id, mask\_channel + 1, mask2poly(new\_mask[:, :, 0], threashold, x\_scaler, y\_scaler))]

    mask\_channel = 1

    result += [(image\_id, mask\_channel + 1, mask2poly(new\_mask[:, :, 1], threashold, x\_scaler, y\_scaler))]

submission = pd.DataFrame(result, columns=['ImageId', 'ClassType', 'MultipolygonWKT'])

sample = sample.drop('MultipolygonWKT', 1)

submission = sample.merge(submission, on=['ImageId', 'ClassType'], how='left').fillna('MULTIPOLYGON EMPTY')

submission.to\_csv('temp\_b\_s.csv', index=False)

## Additional Example of Start of Pytorch Model

from \_\_future\_\_ import division

import numpy as np

import torch

import torch.nn as nn

import keras

from keras.utils import Sequence

from keras.layers import concatenate, Conv2D, MaxPooling2D, UpSampling2D, Cropping2D, BatchNormalization

from keras import backend as K

import h5py

from keras.optimizers import Nadam

from keras.callbacks import ModelCheckpoint

from keras.backend import binary\_crossentropy

import datetime

import os

import random

import matplotlib.pyplot as plt

import torch.optim as optim

from torch.optim import lr\_scheduler

import time

import copy

img\_rows = 112

img\_cols = 112

smooth = 1e-12

num\_channels = 22

num\_mask\_channels = 2

#Keeping original Jaccard coef code for testing

def jaccard\_coef(y\_true, y\_pred):

    intersection = K.sum(y\_true \* y\_pred, axis=[0, 1, 2])

    sum\_ = K.sum(y\_true + y\_pred, axis=[0, 1, 2])

    jac = (intersection + smooth) / (sum\_ - intersection + smooth)

    return K.mean(jac)

def jaccard\_coef\_int(y\_true, y\_pred):

    y\_pred\_pos = K.round(K.clip(y\_pred, 0, 1))

    intersection = K.sum(y\_true \* y\_pred\_pos, axis=[0, 1, 2])

    sum\_ = K.sum(y\_true + y\_pred\_pos, axis=[0, 1, 2])

    jac = (intersection + smooth) / (sum\_ - intersection + smooth)

    return K.mean(jac)

def jaccard\_coef\_loss(y\_true, y\_pred):

    return -K.log(jaccard\_coef(y\_true, y\_pred)) + binary\_crossentropy(y\_pred, y\_true)

#Jaccard coef defined for Pytorch with help from https://github.com/pytorch/ignite/blob/master/ignite/metrics/confusion\_matrix.py#L129

import numbers

from typing import Optional, Union, Any, Callable, Sequence

from ignite.metrics import Metric, MetricsLambda

from ignite.exceptions import NotComputableError

from ignite.metrics.metric import sync\_all\_reduce, reinit\_\_is\_reduced

\_\_all\_\_ = ["ConfusionMatrix", "mIoU", "IoU", "DiceCoefficient", "cmAccuracy", "cmPrecision", "cmRecall"]

class ConfusionMatrix(Metric):

    """Calculates confusion matrix for multi-class data.

    - `update` must receive output of the form `(y\_pred, y)` or `{'y\_pred': y\_pred, 'y': y}`.

    - `y\_pred` must contain logits and has the following shape (batch\_size, num\_categories, ...)

    - `y` should have the following shape (batch\_size, ...) and contains ground-truth class indices

        with or without the background class. During the computation, argmax of `y\_pred` is taken to determine

        predicted classes.

    Args:

        num\_classes (int): number of classes. See notes for more details.

        average (str, optional): confusion matrix values averaging schema: None, "samples", "recall", "precision".

            Default is None. If `average="samples"` then confusion matrix values are normalized by the number of seen

            samples. If `average="recall"` then confusion matrix values are normalized such that diagonal values

            represent class recalls. If `average="precision"` then confusion matrix values are normalized such that

            diagonal values represent class precisions.

        output\_transform (callable, optional): a callable that is used to transform the

            :class:`~ignite.engine.Engine`'s `process\_function`'s output into the

            form expected by the metric. This can be useful if, for example, you have a multi-output model and

            you want to compute the metric with respect to one of the outputs.

        device (str of torch.device, optional): device specification in case of distributed computation usage.

            In most of the cases, it can be defined as "cuda:local\_rank" or "cuda"

            if already set `torch.cuda.set\_device(local\_rank)`. By default, if a distributed process group is

            initialized and available, device is set to `cuda`.

    Note:

        In case of the targets `y` in `(batch\_size, ...)` format, target indices between 0 and `num\_classes` only

        contribute to the confusion matrix and others are neglected. For example, if `num\_classes=20` and target index

        equal 255 is encountered, then it is filtered out.

    """

    def \_\_init\_\_(

        self,

        num\_classes: int,

        average: Optional[str] = None,

        output\_transform: Callable = lambda x: x,

        device: Optional[Union[str, torch.device]] = None,

    ):

        if average is not None and average not in ("samples", "recall", "precision"):

            raise ValueError("Argument average can None or one of ['samples', 'recall', 'precision']")

        self.num\_classes = num\_classes

        self.\_num\_examples = 0

        self.average = average

        self.confusion\_matrix = None

        super(ConfusionMatrix, self).\_\_init\_\_(output\_transform=output\_transform, device=device)

    @reinit\_\_is\_reduced

    def reset(self) -> None:

        self.confusion\_matrix = torch.zeros(self.num\_classes, self.num\_classes, dtype=torch.int64, device=self.\_device)

        self.\_num\_examples = 0

    def \_check\_shape(self, output: Sequence[torch.Tensor]) -> None:

        y\_pred, y = output

        if y\_pred.ndimension() < 2:

            raise ValueError(

                "y\_pred must have shape (batch\_size, num\_categories, ...), " "but given {}".format(y\_pred.shape)

            )

        if y\_pred.shape[1] != self.num\_classes:

            raise ValueError(

                "y\_pred does not have correct number of categories: {} vs {}".format(y\_pred.shape[1], self.num\_classes)

            )

        if not (y.ndimension() + 1 == y\_pred.ndimension()):

            raise ValueError(

                "y\_pred must have shape (batch\_size, num\_categories, ...) and y must have "

                "shape of (batch\_size, ...), "

                "but given {} vs {}.".format(y.shape, y\_pred.shape)

            )

        y\_shape = y.shape

        y\_pred\_shape = y\_pred.shape

        if y.ndimension() + 1 == y\_pred.ndimension():

            y\_pred\_shape = (y\_pred\_shape[0],) + y\_pred\_shape[2:]

        if y\_shape != y\_pred\_shape:

            raise ValueError("y and y\_pred must have compatible shapes.")

    @reinit\_\_is\_reduced

    def update(self, output: Sequence[torch.Tensor]) -> None:

        self.\_check\_shape(output)

        y\_pred, y = output

        self.\_num\_examples += y\_pred.shape[0]

        # target is (batch\_size, ...)

        y\_pred = torch.argmax(y\_pred, dim=1).flatten()

        y = y.flatten()

        target\_mask = (y >= 0) & (y < self.num\_classes)

        y = y[target\_mask]

        y\_pred = y\_pred[target\_mask]

        indices = self.num\_classes \* y + y\_pred

        m = torch.bincount(indices, minlength=self.num\_classes \*\* 2).reshape(self.num\_classes, self.num\_classes)

        self.confusion\_matrix += m.to(self.confusion\_matrix)

    @sync\_all\_reduce("confusion\_matrix", "\_num\_examples")

    def compute(self) -> torch.Tensor:

        if self.\_num\_examples == 0:

            raise NotComputableError("Confusion matrix must have at least one example before it can be computed.")

        if self.average:

            self.confusion\_matrix = self.confusion\_matrix.float()

            if self.average == "samples":

                return self.confusion\_matrix / self.\_num\_examples

            elif self.average == "recall":

                return self.confusion\_matrix / (self.confusion\_matrix.sum(dim=1).unsqueeze(1) + 1e-15)

            elif self.average == "precision":

                return self.confusion\_matrix / (self.confusion\_matrix.sum(dim=0) + 1e-15)

        return self.confusion\_matrix

#This definition calculates the Jaccard index

def IoU(cm: ConfusionMatrix, ignore\_index: Optional[int] = None) -> MetricsLambda:

    """Calculates Intersection over Union using :class:`~ignite.metrics.ConfusionMatrix` metric.

    Args:

        cm (ConfusionMatrix): instance of confusion matrix metric

        ignore\_index (int, optional): index to ignore, e.g. background index

    Returns:

        MetricsLambda

    Examples:

    .. code-block:: python

        train\_evaluator = ...

        cm = ConfusionMatrix(num\_classes=num\_classes)

        IoU(cm, ignore\_index=0).attach(train\_evaluator, 'IoU')

        state = train\_evaluator.run(train\_dataset)

        # state.metrics['IoU'] -> tensor of shape (num\_classes - 1, )

    """

    if not isinstance(cm, ConfusionMatrix):

        raise TypeError("Argument cm should be instance of ConfusionMatrix, but given {}".format(type(cm)))

    if ignore\_index is not None:

        if not (isinstance(ignore\_index, numbers.Integral) and 0 <= ignore\_index < cm.num\_classes):

            raise ValueError("ignore\_index should be non-negative integer, but given {}".format(ignore\_index))

    # Increase floating point precision and pass to CPU

    cm = cm.type(torch.DoubleTensor)

    iou = cm.diag() / (cm.sum(dim=1) + cm.sum(dim=0) - cm.diag() + 1e-15)

    if ignore\_index is not None:

        def ignore\_index\_fn(iou\_vector):

            if ignore\_index >= len(iou\_vector):

                raise ValueError(

                    "ignore\_index {} is larger than the length of IoU vector {}".format(ignore\_index, len(iou\_vector))

                )

            indices = list(range(len(iou\_vector)))

            indices.remove(ignore\_index)

            return iou\_vector[indices]

        return MetricsLambda(ignore\_index\_fn, iou)

    else:

        return iou

#U-Net in pytorch modified from https://github.com/usuyama/pytorch-unet/blob/master/pytorch\_unet.py

def double\_conv(in\_channels, out\_channels):

    return nn.Sequential(

        nn.Conv2d(in\_channels, out\_channels, 3, padding=1),

        nn.ELU(inplace=True),

        nn.Conv2d(out\_channels, out\_channels, 3, padding=1),

        nn.ELU(inplace=True)

    )

class UNet(nn.Module):

    def \_\_init\_\_(self, n\_class):

        super().\_\_init\_\_()

        self.dconv\_down0 = double\_conv(3, 32)

        self.dconv\_down1 = double\_conv(32, 64)

        self.dconv\_down2 = double\_conv(64, 128)

        self.dconv\_down3 = double\_conv(128, 256)

        self.dconv\_down4 = double\_conv(256, 512)

        self.maxpool = nn.MaxPool2d(2)

        self.upsample = nn.Upsample(scale\_factor=2, mode='bilinear', align\_corners=True)

        self.dconv\_up3 = double\_conv(256 + 512, 256)

        self.dconv\_up2 = double\_conv(128 + 256, 128)

        self.dconv\_up1 = double\_conv(128 + 64, 64)

        self.dconv\_up0 = double\_conv(64 + 32, 32)

        self.conv\_last = nn.Conv2d(32, n\_class, 1)

    def forward(self, x):

        conv0 = self.dconv\_down0(x)

        x = self.maxpool(conv1)

        conv1 = self.dconv\_down1(x)

        x = self.maxpool(conv1)

        conv2 = self.dconv\_down2(x)

        x = self.maxpool(conv2)

        conv3 = self.dconv\_down3(x)

        x = self.maxpool(conv3)

        x = self.dconv\_down4(x)

        x = self.upsample(x)

        x = torch.cat([x, conv3], dim=1)

        x = self.dconv\_up3(x)

        x = self.upsample(x)

        x = torch.cat([x, conv2], dim=1)

        x = self.dconv\_up2(x)

        x = self.upsample(x)

        x = torch.cat([x, conv1], dim=1)

        x = self.dconv\_up1(x)

        x = self.upsample(x)

        x = torch.cat([x, conv0], dim=1)

        x = self.dconv\_up0(x)

        out = self.conv\_last(x)

        return out

def form\_batch(X, y, batch\_size):

    X\_batch = np.zeros((batch\_size, num\_channels, img\_rows, img\_cols))

    y\_batch = np.zeros((batch\_size, num\_mask\_channels, img\_rows-32, img\_cols-32))

    X\_height = X.shape[2]

    X\_width = X.shape[3]

    for i in range(batch\_size):

        random\_width = random.randint(0, X\_width - img\_cols - 1)

        random\_height = random.randint(0, X\_height - img\_rows - 1)

        random\_image = random.randint(0, X.shape[0] - 1)

        X\_batch[i] = X[random\_image, :, random\_height: random\_height + img\_rows, random\_width: random\_width + img\_cols]

        yb = y[random\_image, :, random\_height: random\_height + img\_rows, random\_width: random\_width + img\_cols]

        y\_batch[i] = yb[:, 16:16 + img\_rows - 32, 16:16 + img\_cols - 32]

    return np.transpose(X\_batch, (0, 2, 3, 1)), np.transpose(y\_batch, (0, 2, 3, 1))

class data\_generator(Sequence):

    def \_\_init\_\_(self, x\_set, y\_set, batch\_size, horizontal\_flip, vertical\_flip, swap\_axis):

        self.swap\_axis = swap\_axis

        self.vertical\_flip = vertical\_flip

        self.horizontal\_flip = horizontal\_flip

        self.x, self.y = x\_set, y\_set

        self.batch\_size = batch\_size

    def \_\_len\_\_(self):

        return int(np.ceil(len(self.x) / float(self.batch\_size)))

    def \_\_getitem\_\_(self, idx):

        X\_batch, y\_batch = form\_batch(self.x, self.y, self.batch\_size)

        for i in range(X\_batch.shape[0]):

            xb = X\_batch[i]

            yb = y\_batch[i]

            if self.horizontal\_flip:

                if np.random.random() < 0.5:

                    xb = np.fliplr(xb)

                    yb = np.fliplr(yb)

            if self.vertical\_flip:

                if np.random.random() < 0.5:

                    xb = np.flipud(xb)

                    yb = np.flipud(yb)

            if self.swap\_axis:

                if np.random.random() < 0.5:

                    xb = np.rot90(xb)

                    yb = np.rot90(yb)

            X\_batch[i] = xb

            y\_batch[i] = yb

        X\_batch = torch.Tensor(X\_batch)

        y\_batch = torch.Tensor(y\_batch)

        return X\_batch, y\_batch #Changed this from yield to return for running the same file and returns tensors for Pytorch loading

if \_\_name\_\_ == '\_\_main\_\_':

    from collections import defaultdict

    import torch.nn.functional as F

    device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

    model = UNet(n\_class=2)

    model = model.to(device)

    def dice\_loss(pred, target, smooth = 1.):

        pred = pred.contiguous()

        target = target.contiguous()

        intersection = (pred \* target).sum(dim=2).sum(dim=2)

        loss = (1 - ((2. \* intersection + smooth) / (pred.sum(dim=2).sum(dim=2) + target.sum(dim=2).sum(dim=2) + smooth)))

        return loss.mean()

    def calc\_loss(pred, target, metrics, bce\_weight=0.5):

        bce = F.binary\_cross\_entropy\_with\_logits(pred, target)

        pred = F.sigmoid(pred)

        dice = dice\_loss(pred, target)

        loss = bce \* bce\_weight + dice \* (1 - bce\_weight)

        metrics['bce'] += bce.data.cpu().numpy() \* target.size(0)

        metrics['dice'] += dice.data.cpu().numpy() \* target.size(0)

        metrics['loss'] += loss.data.cpu().numpy() \* target.size(0)

        return loss

    def print\_metrics(metrics, epoch\_samples, phase):

        outputs = []

        for k in metrics.keys():

            outputs.append("{}: {:4f}".format(k, metrics[k] / epoch\_samples))

        print("{}: {}".format(phase, ", ".join(outputs)))

    def train\_model(model, optimizer, scheduler, num\_epochs=5):

        best\_model\_wts = copy.deepcopy(model.state\_dict())

        best\_loss = 1e10

        for epoch in range(num\_epochs):

            print('Epoch {}/{}'.format(epoch, num\_epochs - 1))

            print('-' \* 10)

            since = time.time()

            # Each epoch has a training and validation phase

            for phase in ['train', 'val']:

                if phase == 'train':

                    scheduler.step()

                    for param\_group in optimizer.param\_groups:

                        print("LR", param\_group['lr'])

                    model.train()  # Set model to training mode

                else:

                    model.eval()   # Set model to evaluate mode

                metrics = defaultdict(float)

                epoch\_samples = 0

                #Takes in random batch data for training

                for inputs, labels in X\_train, y\_train:

                    inputs = X\_train.to(device)

                    labels = y\_train.to(device)

                    # zero the parameter gradients

                    optimizer.zero\_grad()

                    # forward

                    # track history if only in train

                    with torch.set\_grad\_enabled(phase == 'train'):

                        outputs = model(inputs)

                        loss = calc\_loss(outputs, labels, metrics)

                        # backward + optimize only if in training phase

                        if phase == 'train':

                            loss.backward()

                            optimizer.step()

                    # statistics

                    epoch\_samples += inputs.size(0)

                print\_metrics(metrics, epoch\_samples, phase)

                epoch\_loss = metrics['loss'] / epoch\_samples

                # deep copy the model

                if phase == 'val' and epoch\_loss < best\_loss:

                    print("saving best model")

                    best\_loss = epoch\_loss

                    best\_model\_wts = copy.deepcopy(model.state\_dict())

            time\_elapsed = time.time() - since

            print('{:.0f}m {:.0f}s'.format(time\_elapsed // 60, time\_elapsed % 60))

        print('Best val loss: {:4f}'.format(best\_loss))

        # load best model weights

        model.load\_state\_dict(best\_model\_wts)

        return model

    data\_path = os.getcwd()

    now = datetime.datetime.now()

    print('[{}] Creating and compiling model...'.format(str(datetime.datetime.now())))

    device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

    model = UNet(n\_class=2)

    model = model.to(device)

    print('[{}] Reading train...'.format(str(datetime.datetime.now())))

    f = h5py.File(os.path.join(data\_path, 'train\_t\_c.h5'), 'r')

    X\_train = f['train']

    y\_train = np.array(f['train\_mask'])

    X\_train = torch.Tensor(X\_train).to(device)

    y\_train = torch.Tensor(y\_train).to(device)

    train\_ids = np.array(f['train\_ids'])

    optimizer\_ft = optim.Adam(filter(lambda p: p.requires\_grad, model.parameters()), lr=1e-4)

    exp\_lr\_scheduler = lr\_scheduler.StepLR(optimizer\_ft, step\_size=30, gamma=0.1)

    model = train\_model(model, optimizer\_ft, exp\_lr\_scheduler, num\_epochs=6)

    model()

    f.close()

1. Ronneberger, Olaf; Fischer, Philipp; Brox, Thomas (2015). "U-Net: Convolutional Networks for Biomedical Image Segmentation". arXiv:1505.04597. [↑](#footnote-ref-1)
2. Ma, Lei et. al (2017). “A review of supervised object-based land-cover image classification”. *ISPRS Journal of Photogrammetry and Remote Sensing*. <https://doi.org/10.1016/j.isprsjprs.2017.06.001> [↑](#footnote-ref-2)
3. Dhillon, Anamika; Verma, Gyanendra K. (2019). “Convolutional neural network: a review of models, methodologies and applications to object detection”. *Progress in Artificial Intelligence*. <https://doi.org/10.1007/s13748-019-00203-0> [↑](#footnote-ref-3)
4. Ronneberger, Olaf; Fischer, Philipp; Brox, Thomas (2015). [↑](#footnote-ref-4)
5. https://www.kaggle.com/c/dstl-satellite-imagery-feature-detection [↑](#footnote-ref-5)
6. Ronneberger, Olaf; Fischer, Philipp; Brox, Thomas (2015). [↑](#footnote-ref-6)
7. Iglovikov, Vladimir; Mushinskiy, Sergey; Osin, Vladimir (2017). “Satellite Imagery Feature Detection using Deep Convolutional Neural Network: A Kaggle Competition”. arXiv:1706.06169v1 [↑](#footnote-ref-7)
8. Lambda, Harshall (2019). “Understanding Semantic Segmentation with UNET”. Accessed from <https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-6be4f42d4b47> [↑](#footnote-ref-8)
9. Hagan, Martin T. et.al. (2014). “Neural Network Design (2nd Edition)”. [↑](#footnote-ref-9)
10. Lambda, Harshall (2019). [↑](#footnote-ref-10)
11. A great example of the end-to-end U-Net network for feature classification can be found here <https://www.kaggle.com/ceperaang/lb-0-42-ultimate-full-solution-run-on-your-hw> [↑](#footnote-ref-11)
12. Iglovikov, Vladimir; Mushinskiy, Sergey; Osin, Vladimir (2017). [↑](#footnote-ref-12)
13. <https://deepai.org/machine-learning-glossary-and-terms/jaccard-index> [↑](#footnote-ref-13)
14. Iglovikov, Vladimir; Mushinskiy, Sergey; Osin, Vladimir (2017). [↑](#footnote-ref-14)
15. Iglovikov, Vladimir; Mushinskiy, Sergey; Osin, Vladimir (2017). [↑](#footnote-ref-15)
16. *ibid*. [↑](#footnote-ref-16)