Nick Harbeck and Xinyu Yao

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] land use,[2] and general object detection.[3] Developed in 2015, the U-Net is a fully convolutional network (FCN) that performs image segmentation very well.[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] 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

---

[1] Ronneberger, Olaf; Fischer, Philipp; Brox, Thomas (2015). "U-Net: Convolutional Networks for Biomedical Image Segmentation". arXiv:1505.04597.
[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
[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
[4] Ronneberger, Olaf; Fischer, Philipp; Brox, Thomas (2015).
[5] https://www.kaggle.com/c/dstl-satellite-imagery-feature-detection

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:

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]
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]

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]

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] a U-Net can be generally described in a few steps:

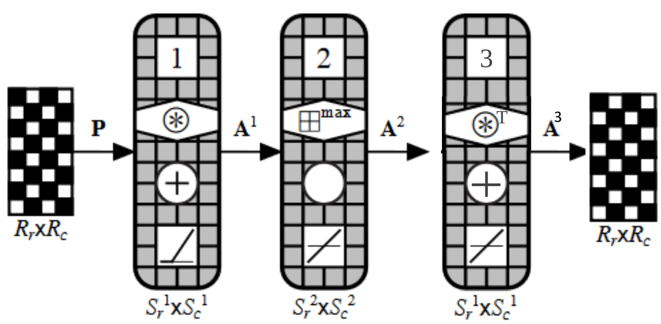

Figure 2 – Generalized Network Representation of a U-Net. The convolution in layer 3 is transposed to up sample the mapped output to the original image size.

---

[6] Ronneberger, Olaf; Fischer, Philipp; Brox, Thomas (2015).

[7] Iglovikov, Vladimir; Mushinskiy, Sergey; Osin, Vladimir (2017). "Satellite Imagery Feature Detection using Deep Convolutional Neural Network: A Kaggle Competition". arXiv:1706.06169v1

[8] Lambda, Harshall (2019). "Understanding Semantic Segmentation with UNET". Accessed from https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-6be4f42d4b47

[9] Hagan, Martin T. et.al. (2014). "Neural Network Design (2nd Edition)".

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]

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] 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')(i
nputs)
    conv1 = BatchNormalization()(conv1)
    conv1 = keras.layers.advanced_activations.ELU()(conv1)
    conv1 = Conv2D(32, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv1)
    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')(u
p9)
    conv9 = BatchNormalization()(conv9)
    conv9 = keras.layers.advanced_activations.ELU()(conv9)
    conv9 = Conv2D(32, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv9)
    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)
```

---

[10] Lambda, Harshall (2019).
[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

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_corner
s=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]

Code to read raw images:

[12] Iglovikov, Vladimir; Mushinskiy, Sergey; Osin, Vladimir (2017).

```
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(imag
e_id)), (1, 2, 0))
    img_p = tiff.imread(data_path + "/sixteen_band/{}_P.tif".format(image_id)).as
type(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.

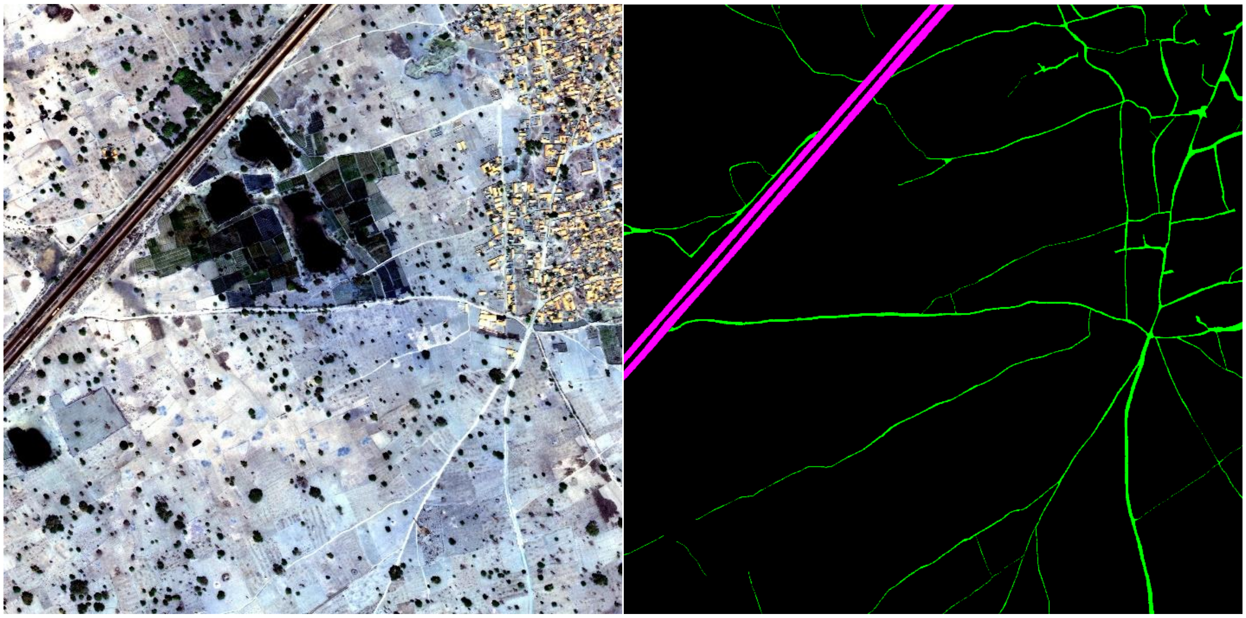

Figure 3 – Example of raw image and its associated mask for the 'roads and tracks' model

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_axi
s=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=Fal
se),
validation_steps = 4,
```

```
callbacks=[ModelCheckpoint(filepath, monitor="val_loss", save_best_only=True, sav
e_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]

[13] https://deepai.org/machine-learning-glossary-and-terms/jaccard-index

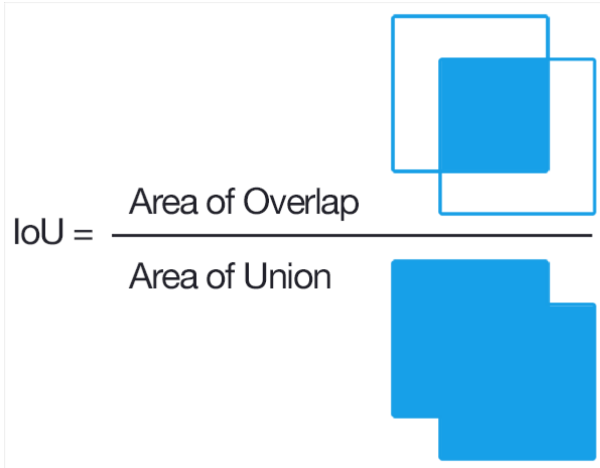

Figure 4 – Visual representation of the Jaccard Index. Image from DeepAI.org.

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)

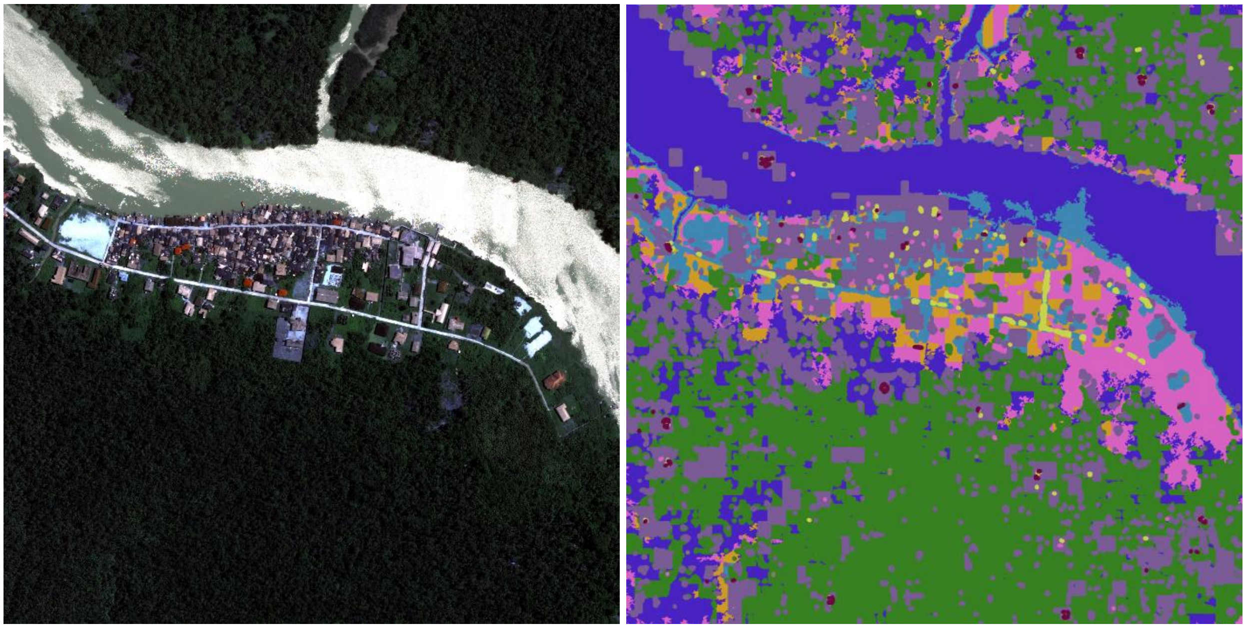

Figure 6 - Example Raw image of 6050_4_4 in the test set (left), and segmented image predicted (right)

Existing research on this dataset found very similar results.[14] 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.

[14] Iglovikov, Vladimir; Mushinskiy, Sergey; Osin, Vladimir (2017).

Roads and Tracks

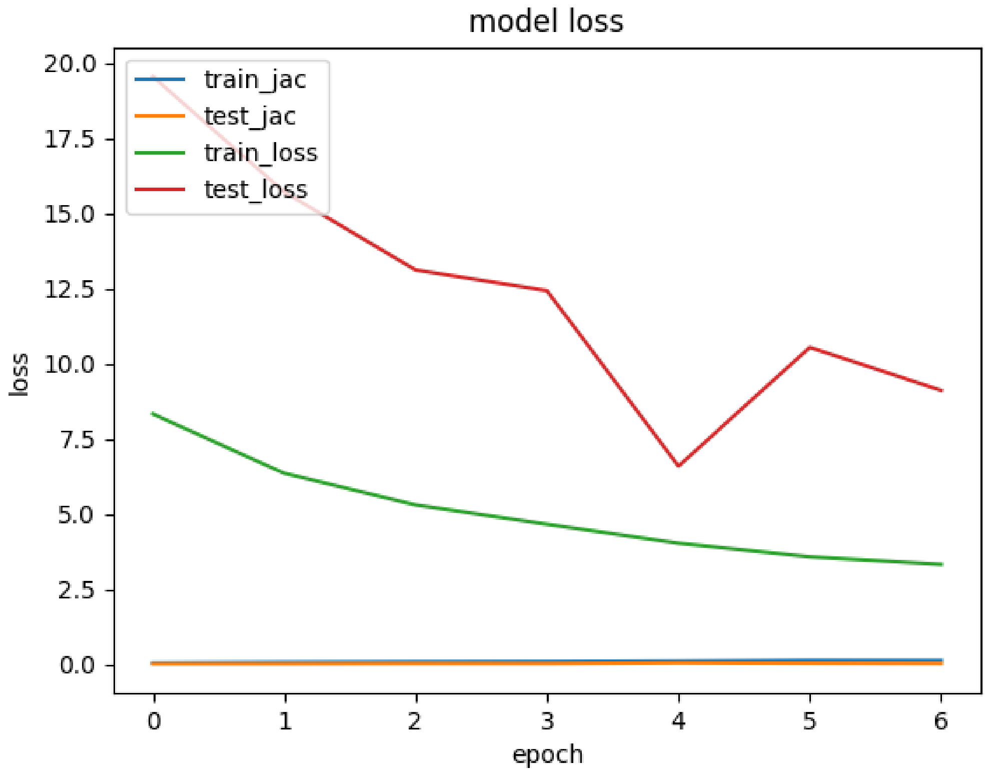

Figure 7 – Graph of model loss at each epoch for training and validation data

Buildings and Misc. Manmade Structures

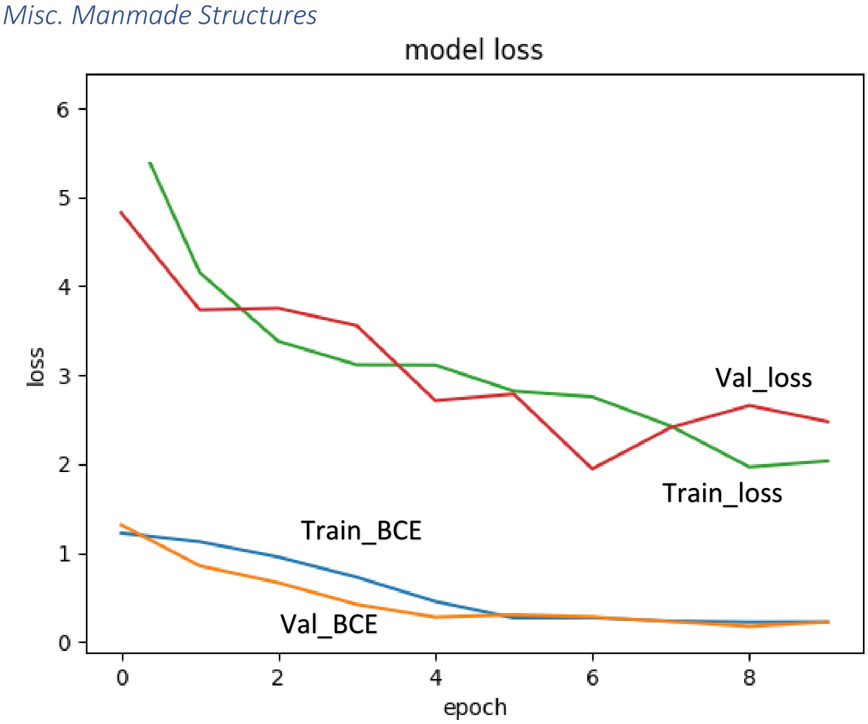

Figure 8 - Graph of model loss at each epoch for training and validation data

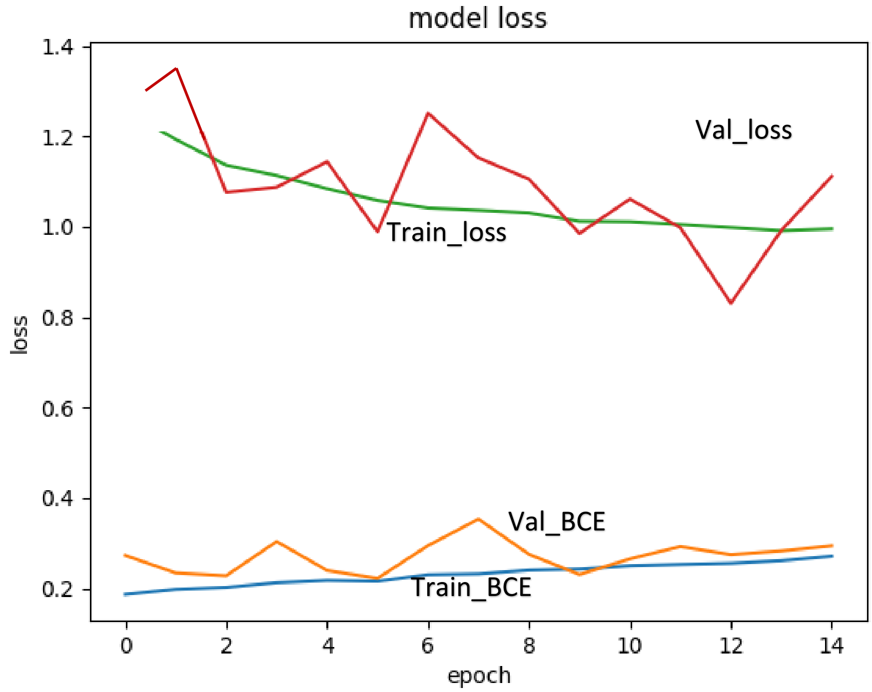

Figure 9 – Example of decreasing Jaccard Index performance with additional epochs

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

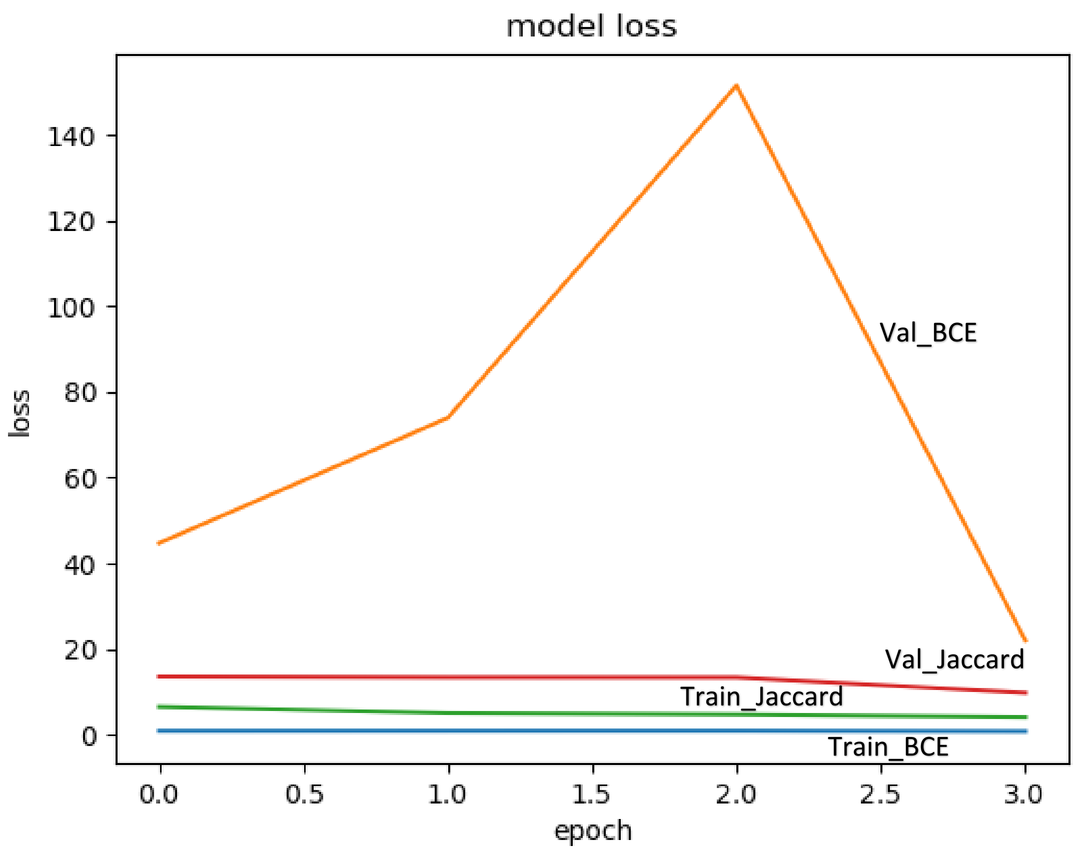

Figure 10 – Graph of model loss at each epoch for training and validation data

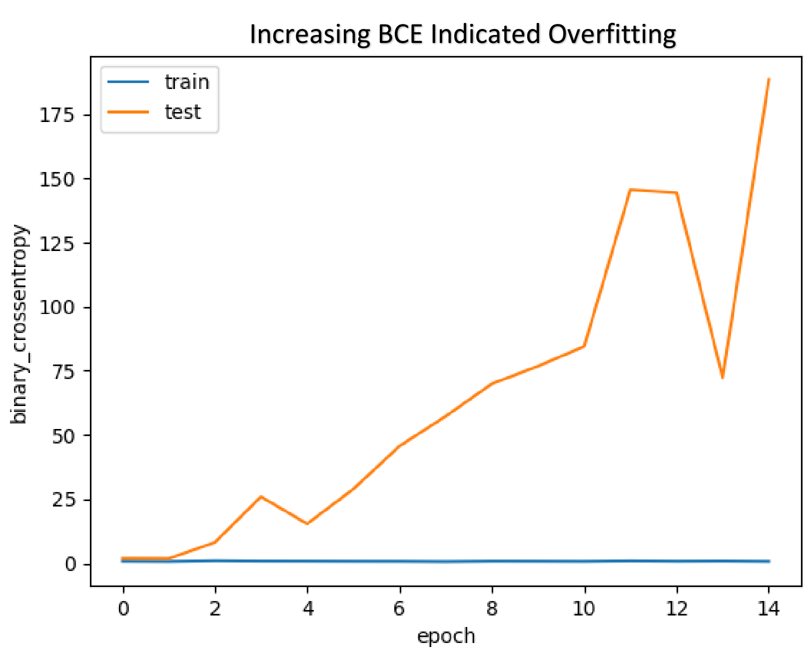

Figure 11 – Example of overfitting indicated by increasing BCE with each epoch

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

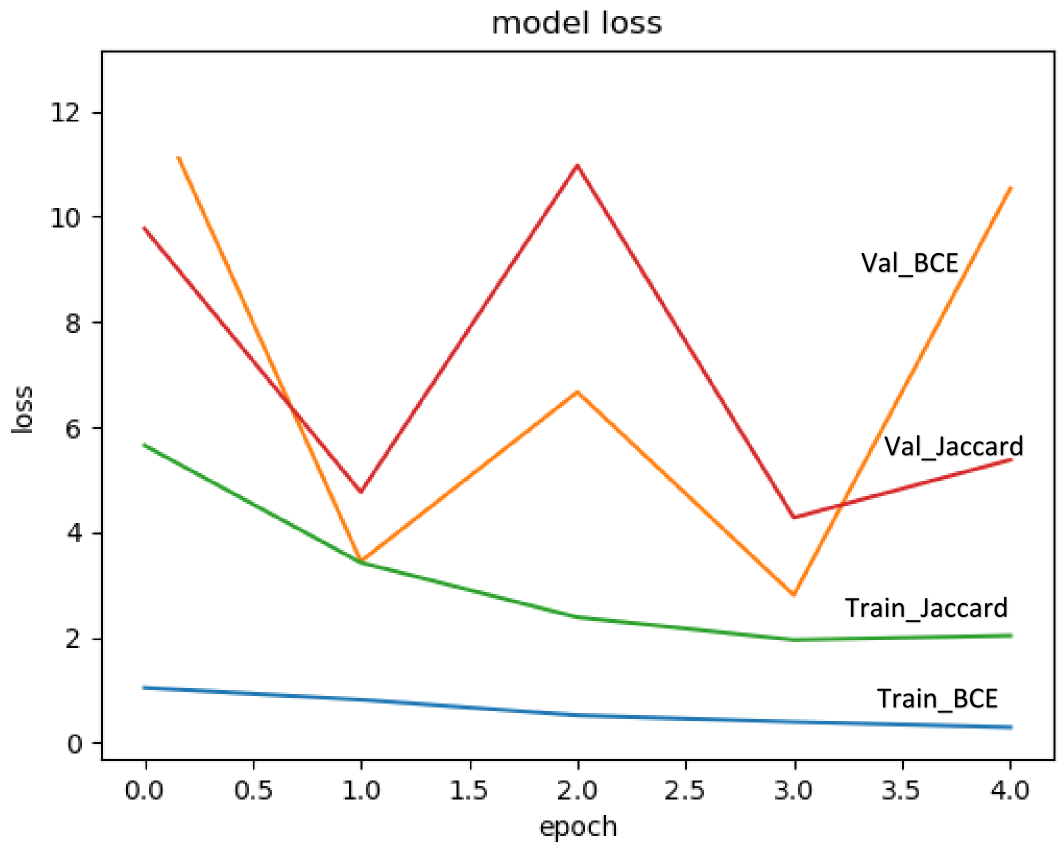

Figure 12 – Graph of model loss at each epoch for training and validation data

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

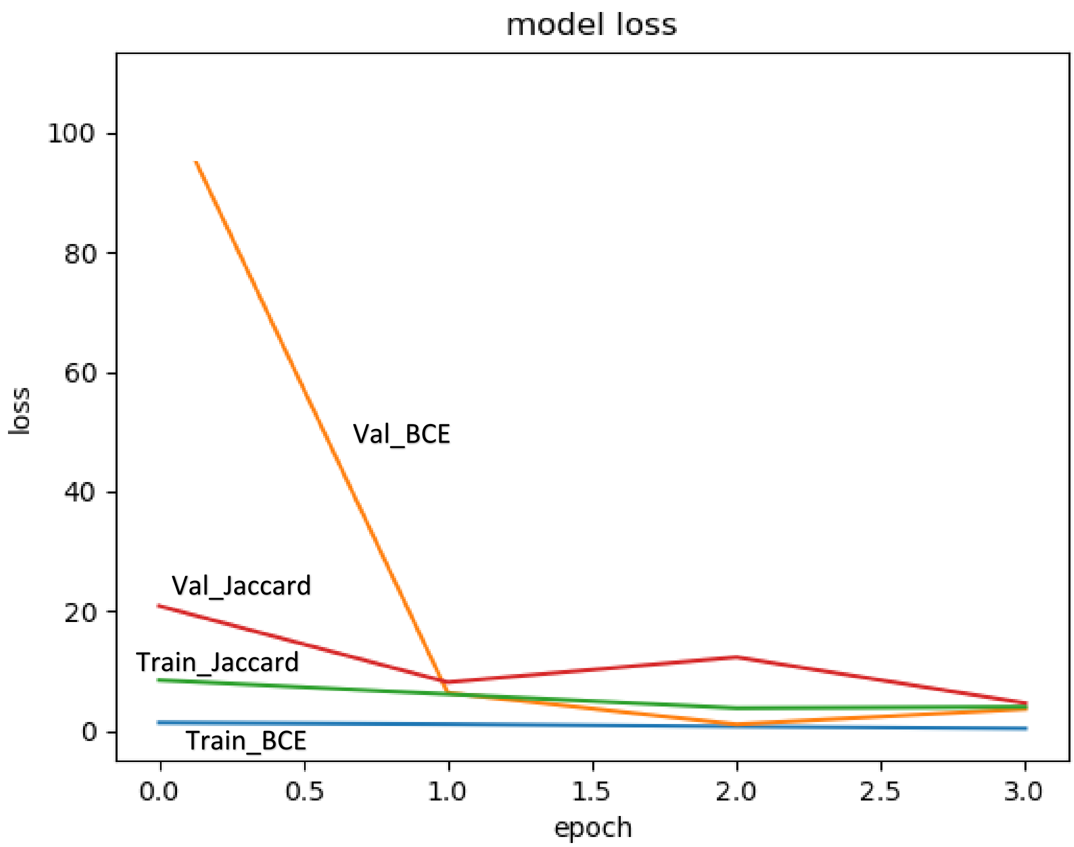

Figure 13 – Graph of model loss at each epoch for training and validation data

There were also some other creative solutions to aid in identifying water that do not rely on a neural network.[15] 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] 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)
```

[15] Iglovikov, Vladimir; Mushinskiy, Sergey; Osin, Vladimir (2017).
[16] *ibid*.

```
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/j
ayspeidell/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', 'Xm
ax', '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, o
rigin=(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.interi
ors]

    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 wi
th 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), 'Mul
tipolygonWKT'].values[0]
        polygons = shapely.wkt.loads(poly)
        mask[mask_channel, :, :] = polygons2mask_layer(height, width, polygons, i
mage_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.contour
Area(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, crit
eria, None, 1)
```

```
    img3 = cv2.warpAffine(img2, warp_matrix, (img1.shape[1], img1.shape[0]), flag
s=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(imag
e_id)), (1, 2, 0))
    img_p = tiff.imread(data_path + "/sixteen_band/{}_P.tif".format(image_id)).as
type(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 pa
rticularly 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 especi
ally important for ecology
    purposes because healthy plants reflect it. Information from this band is imp
ortant 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]] = pred
iction[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 c
reates 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', 'Xm
ax', '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, ima
ge_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), interp
olation=cv2.INTER_CUBIC), (2, 0, 1))
        imgs_mask[i] = np.transpose(
            cv2.resize(np.transpose(extra_functions.generate_mask(image_id, heigh
t, width, start=0,
                                                                    num_mask_channe
ls=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', 'Xm
ax', '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, ima
ge_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), interp
olation=cv2.INTER_CUBIC), (2, 0, 1))
        imgs_mask[i] = np.transpose(
            cv2.resize(np.transpose(extra_functions.generate_mask(image_id, heigh
t, width, start=2,
                                                                   num_mask_channe
ls=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', 'Xm
ax', '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, ima
ge_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), interp
olation=cv2.INTER_CUBIC), (2, 0, 1))
        imgs_mask[i] = np.transpose(
```

```
            cv2.resize(np.transpose(extra_functions.generate_mask(image_id, heigh
t, width, start=4,
                                                                  num_mask_channe
ls=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', 'Xm
ax', '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, ima
ge_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), interp
olation=cv2.INTER_CUBIC), (2, 0, 1))
        imgs_mask[i] = np.transpose(
            cv2.resize(np.transpose(extra_functions.generate_mask(image_id, heigh
t, width, start=0,
                                                                  num_mask_channe
ls=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', 'Xm
ax', '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, ima
ge_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), interp
olation=cv2.INTER_CUBIC), (2, 0, 1))
        imgs_mask[i] = np.transpose(
            cv2.resize(np.transpose(extra_functions.generate_mask(image_id, heigh
t, width, start=6,
                                                                              num_mask_channe
ls=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, Croppin
g2D, 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')(i
nputs)
    conv1 = BatchNormalization()(conv1)
    conv1 = keras.layers.advanced_activations.ELU()(conv1)
    conv1 = Conv2D(32, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv1)
    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')(p
ool1)
    conv2 = BatchNormalization()(conv2)
    conv2 = keras.layers.advanced_activations.ELU()(conv2)
    conv2 = Conv2D(64, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv2)
    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')(u
p8)
    conv8 = BatchNormalization()(conv8)
    conv8 = keras.layers.advanced_activations.ELU()(conv8)
    conv8 = Conv2D(64, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv8)
    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')(u
p9)
    conv9 = BatchNormalization()(conv9)
    conv9 = keras.layers.advanced_activations.ELU()(conv9)
    conv9 = Conv2D(32, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv9)
    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_w
idth: 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, batc
h_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, hor
izontal_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, Croppin
g2D, 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')(i
nputs)
    conv1 = BatchNormalization()(conv1)
    conv1 = keras.layers.advanced_activations.ELU()(conv1)
    conv1 = Conv2D(32, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv1)
    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')(p
ool1)
    conv2 = BatchNormalization()(conv2)
    conv2 = keras.layers.advanced_activations.ELU()(conv2)
    conv2 = Conv2D(64, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv2)
    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')(u
p8)
```

```
    conv8 = BatchNormalization()(conv8)
    conv8 = keras.layers.advanced_activations.ELU()(conv8)
    conv8 = Conv2D(64, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv8)
    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')(u
p9)
    conv9 = BatchNormalization()(conv9)
    conv9 = keras.layers.advanced_activations.ELU()(conv9)
    conv9 = Conv2D(32, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv9)
    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_w
idth: 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 th
e 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, batc
h_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, horizon
tal_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_coe
f_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, Croppin
g2D, 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')(i
nputs)
    conv1 = BatchNormalization()(conv1)
    conv1 = keras.layers.advanced_activations.ELU()(conv1)
    conv1 = Conv2D(32, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv1)
    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')(p
ool1)
    conv2 = BatchNormalization()(conv2)
    conv2 = keras.layers.advanced_activations.ELU()(conv2)
    conv2 = Conv2D(64, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv2)
    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')(u
p8)
    conv8 = BatchNormalization()(conv8)
    conv8 = keras.layers.advanced_activations.ELU()(conv8)
    conv8 = Conv2D(64, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv8)
    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')(u
p9)
    conv9 = BatchNormalization()(conv9)
    conv9 = keras.layers.advanced_activations.ELU()(conv9)
    conv9 = Conv2D(32, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv9)
    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_w
idth: 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 th
e 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, v
ertical_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 th
e 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, batc
h_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, horizon
tal_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_c
oef_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, Croppin
g2D, 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')(i
nputs)
    conv1 = BatchNormalization()(conv1)
    conv1 = keras.layers.advanced_activations.ELU()(conv1)
```

```
    conv1 = Conv2D(32, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv1)
    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')(p
ool1)
    conv2 = BatchNormalization()(conv2)
    conv2 = keras.layers.advanced_activations.ELU()(conv2)
    conv2 = Conv2D(64, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv2)
    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')(u
p8)
    conv8 = BatchNormalization()(conv8)
    conv8 = keras.layers.advanced_activations.ELU()(conv8)
    conv8 = Conv2D(64, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv8)
    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')(u
p9)
    conv9 = BatchNormalization()(conv9)
    conv9 = keras.layers.advanced_activations.ELU()(conv9)
    conv9 = Conv2D(32, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv9)
    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_w
idth: 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 th
e 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 m
odel at least once, it helps to save time in subsequent training steps.
    history = model.fit_generator(generator=data_generator(X_train, y_train, batc
h_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, horizon
tal_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_i
d]['width']), start=0,
                                         num_mask_channels=num_mask_channels, tra
in=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, UpS
ampling2D, 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')(i
nputs)
    conv1 = BatchNormalization()(conv1)
    conv1 = tensorflow.keras.layers.ELU()(conv1)
    conv1 = Conv2D(32, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv1)
    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')(p
ool1)
    conv2 = BatchNormalization()(conv2)
    conv2 = tensorflow.keras.layers.ELU()(conv2)
    conv2 = Conv2D(64, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv2)
```

```
    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')(u
p8)
    conv8 = BatchNormalization()(conv8)
    conv8 = tensorflow.keras.layers.ELU()(conv8)
    conv8 = Conv2D(64, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv8)
    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')(u
p9)
    conv9 = BatchNormalization()(conv9)
    conv9 = tensorflow.keras.layers.ELU()(conv9)
    conv9 = Conv2D(32, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv9)
    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_w
idth: 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_si
ze), batch_size),
                                  epochs=nb_epoch,
                                  verbose=1,
                                  steps_per_epoch=40,
                                  validation_data=gen(batch_size),
                                  validation_steps=4,
                                  callbacks=[ModelCheckpoint(filepath, monitor="v
al_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, UpS
ampling2D, 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')(i
nputs)
    conv1 = BatchNormalization()(conv1)
    conv1 = tensorflow.keras.layers.ELU()(conv1)
    conv1 = Conv2D(32, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv1)
    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')(p
ool1)
    conv2 = BatchNormalization()(conv2)
    conv2 = tensorflow.keras.layers.ELU()(conv2)
    conv2 = Conv2D(64, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv2)
    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')(u
p8)
    conv8 = BatchNormalization()(conv8)
    conv8 = tensorflow.keras.layers.ELU()(conv8)
    conv8 = Conv2D(64, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv8)
    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')(u
p9)
    conv9 = BatchNormalization()(conv9)
    conv9 = tensorflow.keras.layers.ELU()(conv9)
    conv9 = Conv2D(32, (3, 3), padding='same', kernel_initializer='he_uniform')(c
onv9)
    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', 'Xm
ax', '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, e
psilon=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, initia
l_size=(112, 112),
                                                                   final_size=(112-
32, 112-32),
                                                                   num_masks=num_mask_c
hannels, num_channels=num_channels)

    image_v = np.flipud(image)
    predicted_mask_v = extra_functions.make_prediction_cropped(model, image_v, in
itial_size=(112, 112),
                                                                     final_size=(112 -
32, 112 - 32),
                                                                     num_masks=2,
                                                                     num_channels=num_c
hannels)

    image_h = np.fliplr(image)
```

```
    predicted_mask_h = extra_functions.make_prediction_cropped(model, image_h, in
itial_size=(112, 112),
                                                               final_size=(112 -
32, 112 - 32),
                                                               num_masks=2,
                                                               num_channels=num_c
hannels)

    image_s = np.rot90(image)
    predicted_mask_s = extra_functions.make_prediction_cropped(model, image_s, in
itial_size=(112, 112),
                                                               final_size=(112 -
32, 112 - 32),
                                                               num_masks=2,
                                                               num_channels=num_c
hannels)

    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], threasho
ld, x_scaler, y_scaler))]
    mask_channel = 1
    result += [(image_id, mask_channel + 1, mask2poly(new_mask[:, :, 1], threasho
ld, x_scaler, y_scaler))]

submission = pd.DataFrame(result, columns=['ImageId', 'ClassType', 'MultipolygonW
KT'])


sample = sample.drop('MultipolygonWKT', 1)
submission = sample.merge(submission, on=['ImageId', 'ClassType'], how='left').fi
llna('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, Croppin
g2D, 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/ignit
e/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", "cm
Precision", "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_pre
d, 'y': y}`.
    - `y_pred` must contain logits and has the following shape (batch_size, num_c
ategories, ...)
    - `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 norma
lized such that diagonal values
            represent class recalls. If `average="precision"` then confusion matr
ix values are normalized such that
            diagonal values represent class precisions.
        output_transform (callable, optional): a callable that is used to transfo
rm 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 d
istributed 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", "precisio
n"):
            raise ValueError("Argument average can None or one of ['samples', 're
call', '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, d
type=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 {}".for
mat(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 m
ust 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 exa
mple 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.Confusion
Matrix` 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 g
iven {}".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_corner
s=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_w
idth: 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 th
e 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()
```