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

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 25 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:

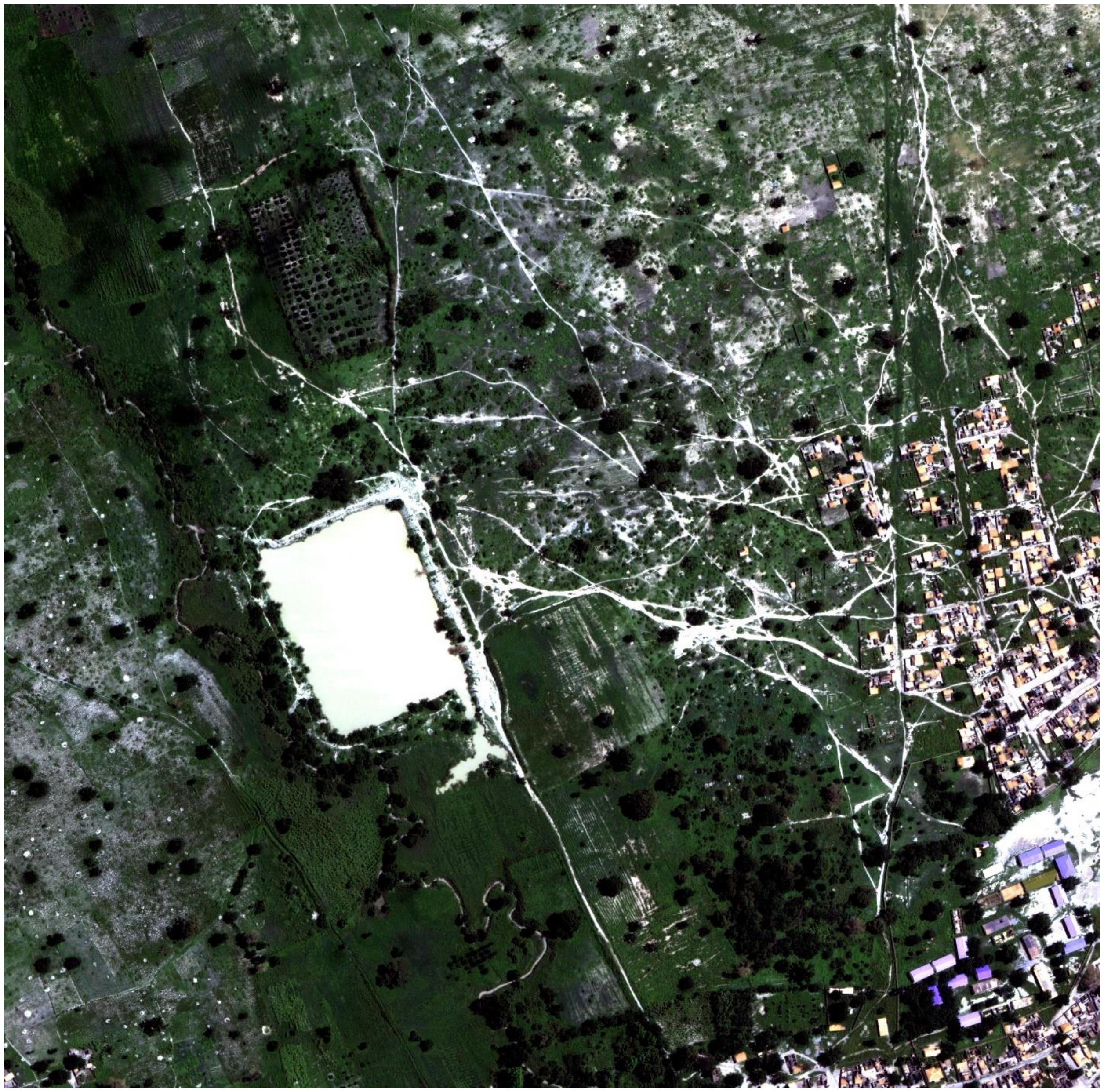

Figure 1 – Example image from the Kaggle DSTL dataset saved as a .png

Description of individual work

For this project, a U-Net was used to perform image segmentation and feature detection.

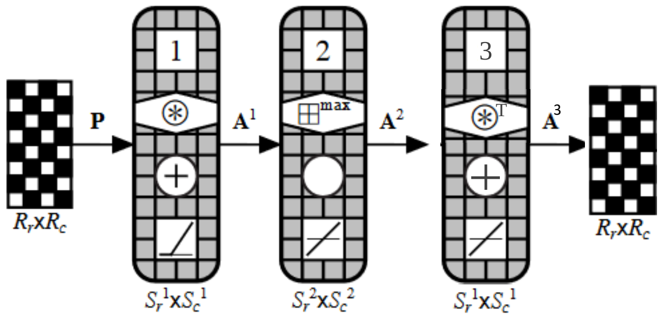

Figure 2 – Generalized Network Representation of a U-Net. The convolution in layer 3 is transposed to up sample the mapped output to the original image size.

The network first convolves over the input data and then pools the output through layers 1 and 2 in the diagram. Next the pooling operates in 'reverse' in layer 3, where the pooled data is resized sequentially to the same dimensions as the original input, sometimes with the help of some padding. Like the kernel of a convolution layer, the transposed convolutions that up sample the inputs are learned.[6]

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.[7] The model has 10 layers and incorporates normalization, Exponential Linear Unit (ELU) activations, pooling, and cropping to achieve better performance.

```
def get_unet0():
    inputs = keras.Input((img_rows, img_cols, num_channels))
    conv1 = Conv2D(32, (3, 3), padding='same',
kernel_initializer='he_uniform')(inputs)
    conv1 = BatchNormalization()(conv1)
    conv1 = keras.layers.advanced_activations.ELU()(conv1)
    conv1 = Conv2D(32, (3, 3), padding='same',
kernel_initializer='he_uniform')(conv1)
    conv1 = BatchNormalization()(conv1)
    conv1 = keras.layers.advanced_activations.ELU()(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)

    conv2 = Conv2D(64, (3, 3), padding='same',
kernel_initializer='he_uniform')(pool1)
```

[6] Lambda, Harshall (2019).
[7] 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

```
    conv2 = BatchNormalization()(conv2)
    conv2 = keras.layers.advanced_activations.ELU()(conv2)
    conv2 = Conv2D(64, (3, 3), padding='same',
kernel_initializer='he_uniform')(conv2)
    conv2 = BatchNormalization()(conv2)
    conv2 = keras.layers.advanced_activations.ELU()(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)

    conv3 = Conv2D(128, (3, 3), padding='same',
kernel_initializer='he_uniform')(pool2)
    conv3 = BatchNormalization()(conv3)
    conv3 = keras.layers.advanced_activations.ELU()(conv3)
    conv3 = Conv2D(128, (3, 3), padding='same',
kernel_initializer='he_uniform')(conv3)
    conv3 = BatchNormalization()(conv3)
    conv3 = keras.layers.advanced_activations.ELU()(conv3)
    pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)

    conv4 = Conv2D(256, (3, 3), padding='same',
kernel_initializer='he_uniform')(pool3)
    conv4 = BatchNormalization()(conv4)
    conv4 = keras.layers.advanced_activations.ELU()(conv4)
    conv4 = Conv2D(256, (3, 3), padding='same',
kernel_initializer='he_uniform')(conv4)
    conv4 = BatchNormalization()(conv4)
    conv4 = keras.layers.advanced_activations.ELU()(conv4)
    pool4 = MaxPooling2D(pool_size=(2, 2))(conv4)

    conv5 = Conv2D(512, (3, 3), padding='same',
kernel_initializer='he_uniform')(pool4)
    conv5 = BatchNormalization()(conv5)
    conv5 = keras.layers.advanced_activations.ELU()(conv5)
    conv5 = Conv2D(512, (3, 3), padding='same',
kernel_initializer='he_uniform')(conv5)
    conv5 = BatchNormalization()(conv5)
    conv5 = keras.layers.advanced_activations.ELU()(conv5)

    up6 = concatenate([UpSampling2D(size=(2, 2))(conv5), conv4], axis=3)
    conv6 = Conv2D(256, (3, 3), padding='same',
kernel_initializer='he_uniform')(up6)
    conv6 = BatchNormalization()(conv6)
    conv6 = keras.layers.advanced_activations.ELU()(conv6)
    conv6 = Conv2D(256, (3, 3), padding='same',
kernel_initializer='he_uniform')(conv6)
    conv6 = BatchNormalization()(conv6)
    conv6 = keras.layers.advanced_activations.ELU()(conv6)

    up7 = concatenate([UpSampling2D(size=(2, 2))(conv6), conv3], axis=3)
    conv7 = Conv2D(128, (3, 3), padding='same',
kernel_initializer='he_uniform')(up7)
    conv7 = BatchNormalization()(conv7)
    conv7 = keras.layers.advanced_activations.ELU()(conv7)
    conv7 = Conv2D(128, (3, 3), padding='same',
kernel_initializer='he_uniform')(conv7)
    conv7 = BatchNormalization()(conv7)
    conv7 = keras.layers.advanced_activations.ELU()(conv7)

    up8 = concatenate([UpSampling2D(size=(2, 2))(conv7), conv2], axis=3)
    conv8 = Conv2D(64, (3, 3), padding='same',
kernel_initializer='he_uniform')(up8)
    conv8 = BatchNormalization()(conv8)
    conv8 = keras.layers.advanced_activations.ELU()(conv8)
    conv8 = Conv2D(64, (3, 3), padding='same',
kernel_initializer='he_uniform')(conv8)
    conv8 = BatchNormalization()(conv8)
    conv8 = keras.layers.advanced_activations.ELU()(conv8)

    up9 = concatenate([UpSampling2D(size=(2, 2))(conv8), conv1], axis=3)
    conv9 = Conv2D(32, (3, 3), padding='same',
kernel_initializer='he_uniform')(up9)
    conv9 = BatchNormalization()(conv9)
    conv9 = keras.layers.advanced_activations.ELU()(conv9)
    conv9 = Conv2D(32, (3, 3), padding='same',
kernel_initializer='he_uniform')(conv9)
    crop9 = Cropping2D(cropping=((16, 16), (16, 16)))(conv9)
    conv9 = BatchNormalization()(crop9)
    conv9 = keras.layers.advanced_activations.ELU()(conv9)
    conv10 = Conv2D(num_mask_channels, (1, 1), activation='sigmoid')(conv9)

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

    return model
```

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
4. Choose Nadam as optimizer

```
model.compile(optimizer=Nadam(lr=1e-3), loss=jaccard_coef_loss,
metrics=['binary_crossentropy', jaccard_coef_int])
model.load_weights('b_s.h5')
history = model.fit_generator(generator=data_generator(X_train, y_train,
batch_size, horizontal_flip=True, vertical_flip=True, swap_axis=True),
                    epochs=nb_epoch,
                    verbose=1,
                    samples_per_epoch=batch_size * 400,
                    validation_data=data_generator(X_train, y_train, 128,
horizontal_flip=False, vertical_flip=False, swap_axis=False),
                    validation_steps = 4,
                    callbacks=[ModelCheckpoint(filepath, monitor="val_loss",
save_best_only=True, save_weights_only=True)],
                    workers=8
                    )
```

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.

1. Image pre-processing

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.[8]

1) Resize: 3 bands images and other 16 bands images are in different shapes, therefore, 16 bands images are resized to the shape of 3 bands.
2) Standardize pixels: all the pixels are rescaled to (0,1)
3) Alignment: 3 bands images and other 16 bands images are slightly offset, therefore, alignment was performed to bring 16 bands images with 3 bands.
4) Data augmentation: CCCI and NDWI reflectance indices[12], that defined as follows:

$$\mathrm{CCCI} = \frac{\mathrm{NIR} - \mathrm{RED}_{\mathrm{edge}}}{\mathrm{NIR} + \mathrm{RED}_{\mathrm{edge}}} \times \frac{\mathrm{NIR} + \mathrm{RED}}{\mathrm{NIR} - \mathrm{RED}}$$

$$\mathrm{NDWI} = \frac{\mathrm{GREEN} - \mathrm{NIR}}{\mathrm{GREEN} + \mathrm{NIR}}$$

where both indices are represented as ratios of the difference and sum of pixel values in the green, red edge, red and infrared channels. The result shows high intensity values for waterways, but it also shows false positives for some buildings due to the relative similarity of the specific heat of metal roofs and water.

Code to read raw images:

```
def read_image_22(image_id):
    img_a = np.transpose(tiff.imread(data_path + "/sixteen_band/{}_A.tif".format(
image_id)), (1, 2, 0))
    img_m = np.transpose(tiff.imread(data_path + "/sixteen_band/{}_M.tif".format(
image_id)), (1, 2, 0)) # h w c
    img_3 = np.transpose(tiff.imread(data_path + "/three_band/{}.tif".format(imag
e_id)), (1, 2, 0))
```

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

```
    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 code to standardize pixels:

```
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)
```

The code to align images:

```
def _align_two_rasters(img1,img2, band):
    i=0
    if band == 'A':
        i= 3
    elif band == 'M':
        i = 5
    p1 = img1[:, :, 1]
    p2 = img2[:, :, i]
    warp_mode = cv2.MOTION_EUCLIDEAN
    warp_matrix = np.eye(2, 3, dtype=np.float32)
    criteria = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 1000, 1e-7)
    (cc, warp_matrix) = cv2.findTransformECC (p1, p2,warp_matrix, warp_mode,
criteria, None, 1)
    img3 = cv2.warpAffine(img2, warp_matrix, (img1.shape[1], img1.shape[0]),
flags=cv2.INTER_LINEAR + cv2.WARP_INVERSE_MAP)
    img3[img3 == 0] = np.average(img3)

    return img3
```

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

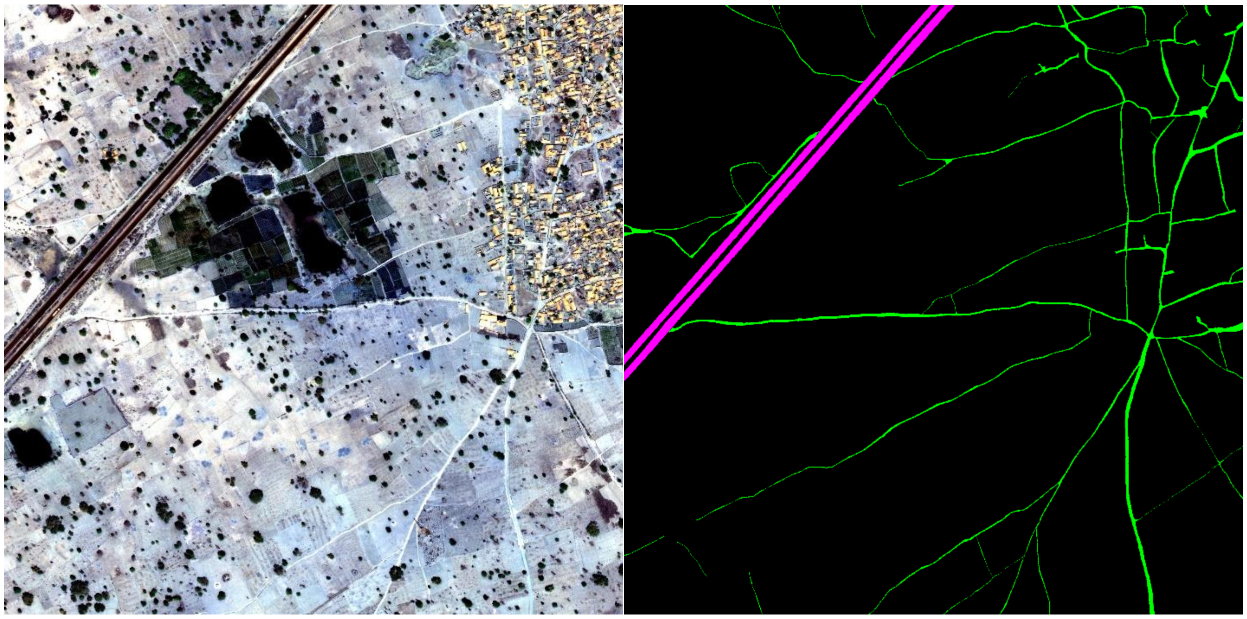

Figure 3 – Example of raw image and its associated mask for the 'roads and tracks' model

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))
```

The loss function we use is `(−log(jaccard) + BCE`, where jaccard score, also known as IoU (interaction over union), and BCE, as binary-cross-entropy.

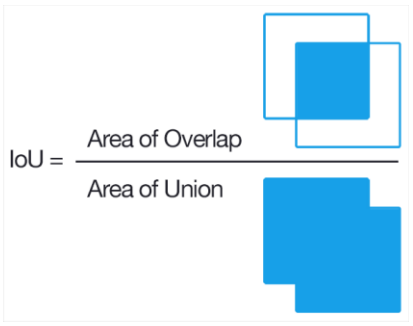

Figure 4 – Visual representation of the Jaccard Index. Image from DeepAI.org.

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.

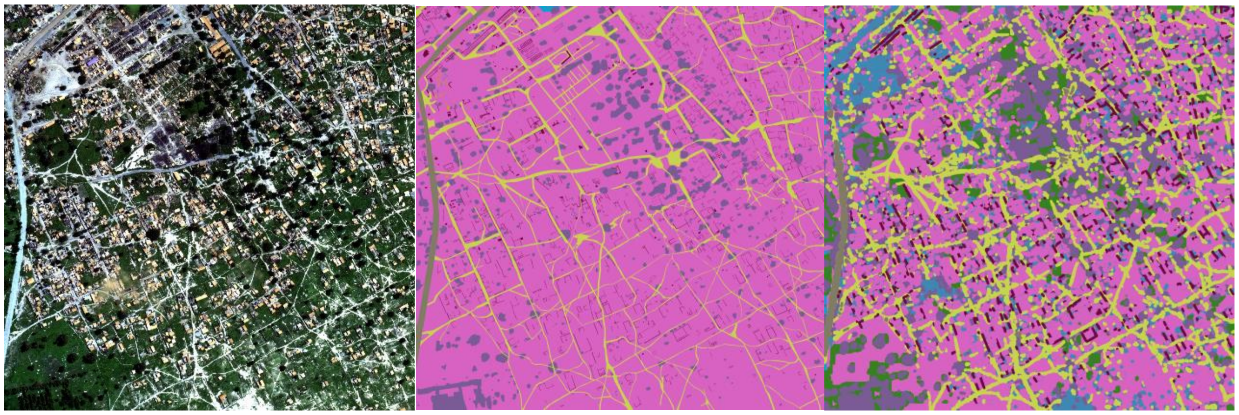

Figure 5 - Example Raw image of 6100_2_3 in the training set (left), mask generated from labeled polygon (middle) and segmented image predicted (right)

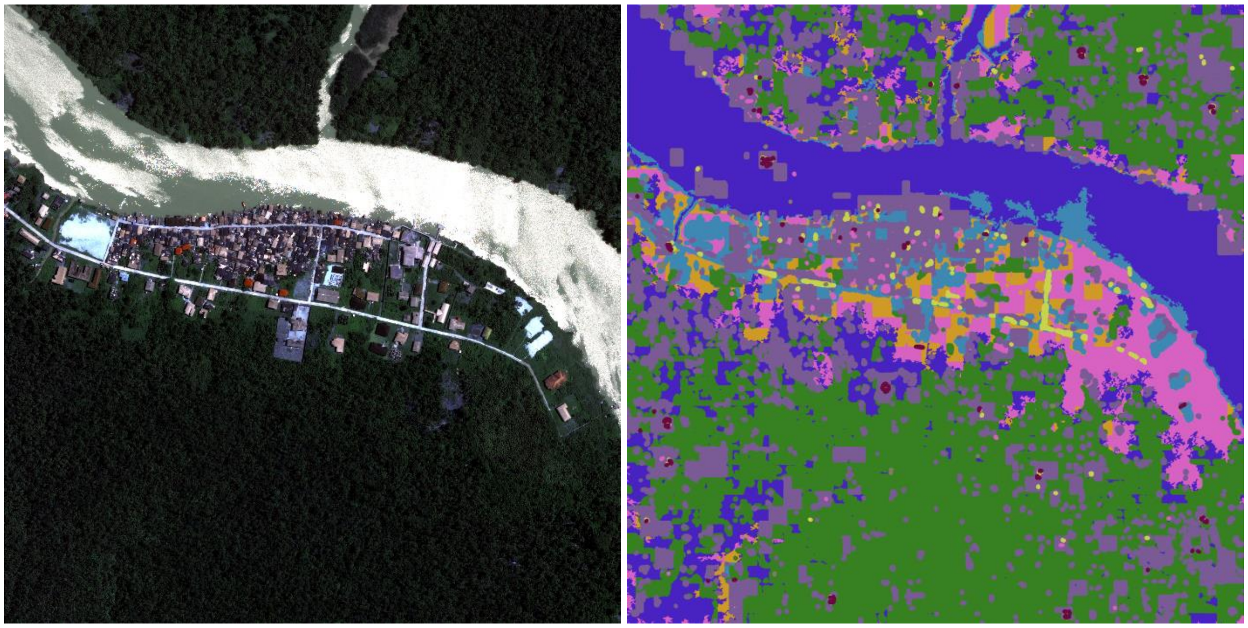

Figure 6 - Example Raw image of 6050_4_4 in the test set (left), and segmented image predicted (right)

Existing research on this dataset found very similar results.[9] 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.

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

Roads and Tracks

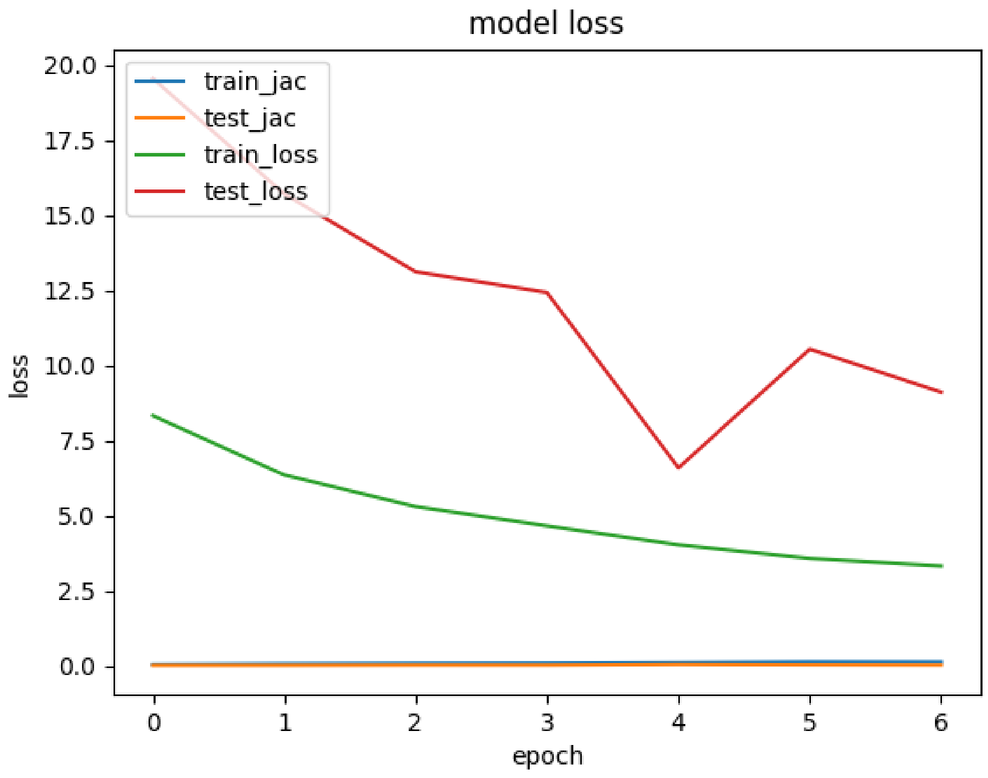

Figure 7 – Graph of model loss at each epoch for training and validation data

Buildings and Misc. Manmade Structures

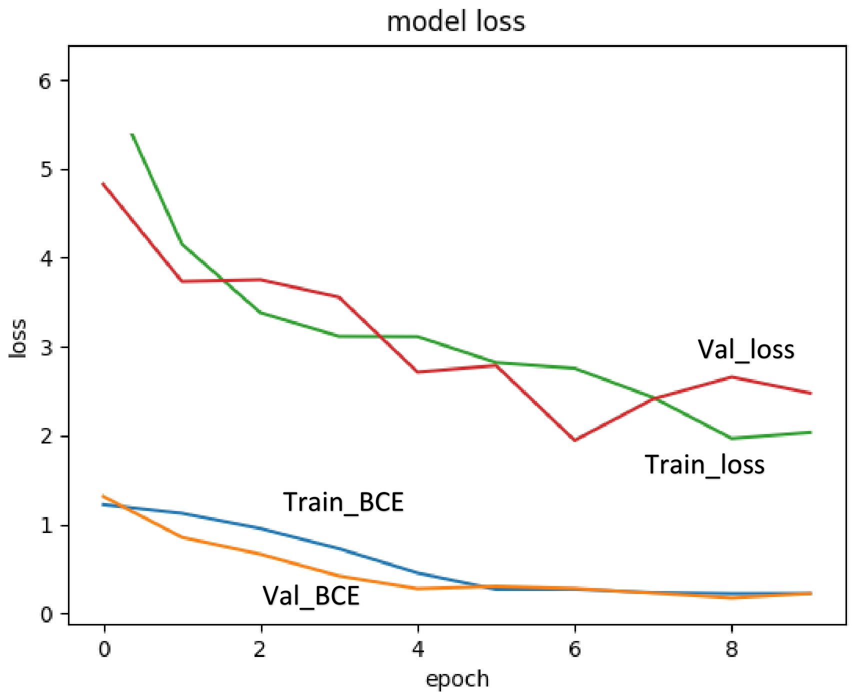

Figure 8 - Graph of model loss at each epoch for training and validation data

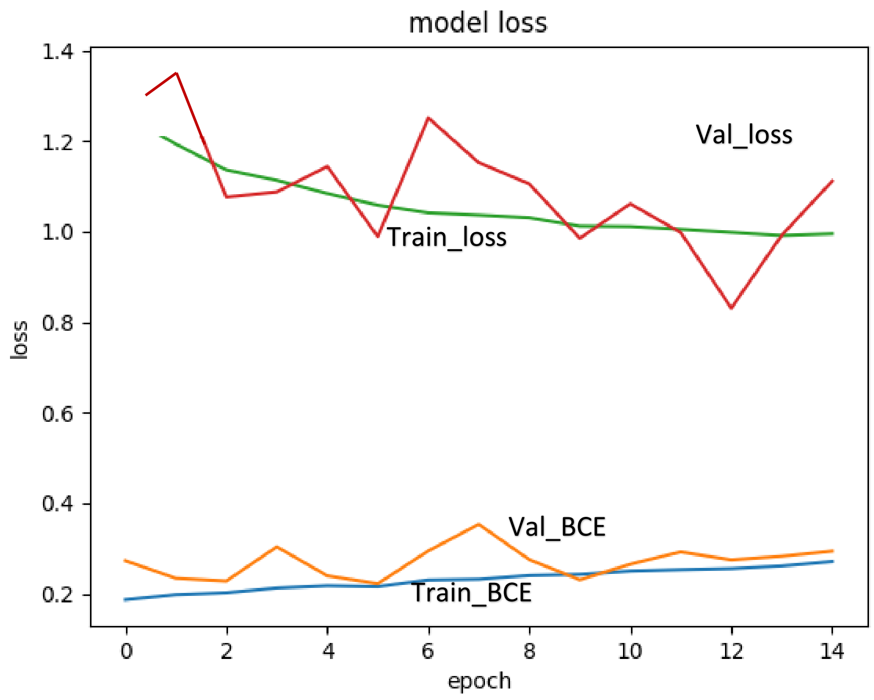

Figure 9 – Example of decreasing Jaccard Index performance with additional epochs

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.

There were also some other creative solutions to aid in identifying water that do not rely on a neural network.[10] 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[11] and could be helpful for further distinguishing water from other parts of this dataset or future datasets.

Example code using the CCCI to distinguish water:

```
def mask2poly_fastwater(predicted_mask, x_scaler, y_scaler):
    polygons = extra_functions.mask2polygons_layer(predicted_mask, epsilon=0, 
min_area=10000)
    polygons = shapely.affinity.scale(polygons, xfact=1.0 / x_scaler, yfact=1.0 / 
y_scaler, origin=(0, 0, 0))
    return shapely.wkt.dumps(polygons)

def mask2poly_slowwater(predicted_mask, x_scaler, y_scaler):
    polygons = extra_functions.mask2polygons_layer(predicted_mask, epsilon=0, 
min_area=1000)
```

[10] Iglovikov, Vladimir; Mushinskiy, Sergey; Osin, Vladimir (2017).
[11] *ibid*.

```
    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))]
```

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.

Calculation

Numerator (41 - 4) + 204 – 22 + 21 – 1 + 108 – 8 + 147 – 7 + 166 -16 + 144 - 14 = 759
Denominator 41 + 204 + 8 + 21 + 20 + 108 + 147 + 166 + 26 + 144 +14 = 899
Percentage 789 / 899 = 84%

References

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

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