DSTL Satellite Imagery Feature Detection - Code Report

May 12, 2019

Group 22: Alex Spiride, Charles Flood, Daniel Varon, and Dillon Smith
View our Ignite Presentation here: https://tinyurl.com/CS109b-Group22

Contents

1 Introduction

As the size of satellite imagery datasets continues to increase so does the need for automated classification. In 2017, the Defence Science and Technology Laboratory (DSTL) held a Kaggle competition focused on classifying surface features in hyperspectral imagery of the Earth's surface from images captured by the Worldview-3 satellite instruments. The task was to train a model that could accurately distinguish between 10 different surface classes: buildings, miscellaneous human-made structures, roads, tracks, trees, crops, waterways, standing water, large vehicles, and small vehicles. For the competition, DSTL provided 20GB of labeled surface imagery in 16 spectral bands from 400-2365 nm and with 0.31-7.5 m pixel resolution. We propose to help solve this segmentation problem using fully convolutional neural networks with the U-net architecture developed by Ronneberger et al,[1] while incorporating additional methods that were not present in most of the other Kaggle submissions that involved U-nets. These include decaying learning rates, image sub-setting and varying the number of input channels for different classes.

The DSTL hoped that through the results of the competition, the possibility of automated feature labeling would "not only help DSTL make smart decisions more quickly around the defense and security of the UK, but also bring innovation to computer vision methodologies applied to satellite imagery."[2]

DSTL provided 1 square kilometer of satellite images in both 3-band (RGB) and 16-band formats from the Worldview 3 satellite. More details about these images are below.[3]

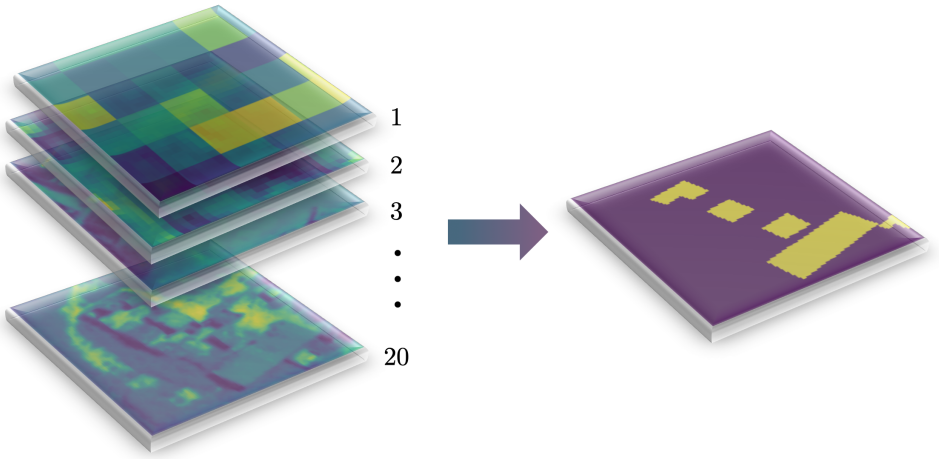
- Wavebands :
  - Panchromatic: 450-800 nm
  - 8 Multispectral: (red, red edge, coastal, blue, green, yellow, near-IR1 and near-IR2) 400 nm - 1040 nm
  - 8 SWIR: 1195 nm - 2365 nm
- Sensor Resolution (GSD) at Nadir :
  - Panchromatic: 0.31m
  - Multispectral: 1.24 m
  - SWIR: Delivered at 7.5m
- Dynamic Range
  - Panchromatic and multispectral : 11-bits per pixel
  - SWIR : 14-bits per pixel

[1] https://arxiv.org/pdf/1505.04597.pdf
[2] https://www.kaggle.com/c/dstl-satellite-imagery-feature-detection/data
[3] https://www.kaggle.com/c/dstl-satellite-imagery-feature-detection/data

2 Data Preprocessing

The data preprocessing for the project consisted of three main parts: preprocessing the spectral data, preprocessing the labels data, and tiling the data. At the beginning of the process, each satellite image was stored in four separate tiff image files—one capturing natural RGB color, one that is panchromatic, one from the multispectral range (400-1040nm), and one from the short-wave infrared (SWIR) range (1195-2365nm)—and masks are stored as WKT multipolygons. Our

goal was to use this data for each image and turn it into a datacube containing each of the tiff image files that could be tiled with the masks to create smaller cubes which would be fed into our model at the end of the preprocessing.



The following three code chunks import the necessary packages and functions for the preprocessing, set the global parameters for the preprocessing, and set up the data for the preprocessing, respectively.

```
In [ ]: import cv2
        from shapely.wkt import loads as wkt_loads
        import rasterio
        import numpy as np
        import os
        import shapely
        from rasterio import features
        import shapely.geometry
        import shapely.affinity
        import pandas as pd
        import tifffile as tiff
        from collections import Counter
        import matplotlib.pyplot as plt
        import pickle
        from scipy.interpolate import interp2d
        %matplotlib inline
        from IPython.core.display import Image, display
```

```
In [ ]: CLASS_TYPES = ['buildings',                  # These are the different class types,
                                                    # 1 through ...
                      'misc_structures',            # 2
                      'road',                       # 3
                      'track',                      # 4
                      'trees',                      # 5
                      'crops',                      # 6
                      'waterway',                   # 7
                      'standing_water',             # 8
```

```
                'vehicle_small',                # 9
                'vehicle_large']                # 10
TILE_SIZE = 128                                 # Size of mini-tiles for subsampling
                                                # the data
FEATURE = 'buildings'                           # Surface feature of interest - data
                                                # will be
                                                # made for this feature
MIN_CLASSES = 7                                 # Only looking at images with this
                                                # many classes
DATA_DIR = 'data/data_tiles'                    # Data will be saved here
```

```
In [ ]: polydf = pd.read_csv('data/train_wkt_v4.csv')
        train_images = polydf.loc[polydf['MultipolygonWKT'] != 'MULTIPOLYGON EMPTY']
        train_images = train_images.groupby('ImageId')['ClassType'].apply(list)
        train_images = train_images.to_frame()
        train_images.reset_index(level=0, inplace=True)
        train_images.columns = ['ImageId', 'ClassTypes']
        train_images['ClassCount'] = train_images.apply(lambda x: len(x['ClassTypes']),
                                                        axis=1)
        train_images = train_images.sort_values('ClassCount', ascending=False)
        train_images.reset_index(level=0, inplace=True)
        train_images = train_images.drop(['index'], axis=1)
```

2.1 Spectral Data

In order to preprocess the spectral data, the first thing we had to do was make each of the image files the same size. Because the RGB and panchromatic images have a resolution of 0.31 m while the multispectral images have a resolution of 1.24 m and the SWIR images have a resolution of 7.5 m. The images also have a different number of layers: each panchromatic image has one, each RGB image has three, each multispectral image has eight, and each SWIR image has eight.

In order to address this issue, we decided to perform interpolation on the images to make them all the same size. For each datacube, we decided to set the panchromatic data as the index file for the size of the data cube. Because multispectral and SWIR images were smaller than the panchromatic images, we used nearest neighbor interpolation to make them the same size. Since the RGB images were, at times, larger than the panchromatic data, we used linear interpolation to make them the same size.

In the code below, we define the nearest neighbor interpolation function (found online at https://gist.github.com/KeremTurgutlu/68feb119c9dd148285be2e247267a203) and linear interpolation function as well as the functions that allow us to access individual layers of a tiff file and create the final (20 layer x ~3000 pixel x ~3000 pixel) datacubes.

```
In [ ]: def nn_interpolate(A, new_size):

            """
            Nearest Neighbor Interpolation, Step by Step
            (code borrowed from
            https://gist.github.com/KeremTurgutlu/68feb119c9dd148285be2e247267a203)
```

```
    """

    # get sizes
    old_size = A.shape

    # calculate row and column ratios
    row_ratio, col_ratio = new_size[0]/old_size[0], new_size[1]/old_size[1]

    # define new pixel row position i
    new_row_positions = np.array(range(new_size[0]))+1
    new_col_positions = np.array(range(new_size[1]))+1

    # normalize new row and col positions by ratios
    new_row_positions = new_row_positions / row_ratio
    new_col_positions = new_col_positions / col_ratio

    # apply ceil to normalized new row and col positions
    new_row_positions = np.ceil(new_row_positions)
    new_col_positions = np.ceil(new_col_positions)

    # find how many times to repeat each element
    row_repeats = np.array(list(Counter(new_row_positions).values()))
    col_repeats = np.array(list(Counter(new_col_positions).values()))

    # perform column-wise interpolation on the columns of the matrix
    row_matrix = np.dstack([np.repeat(A[:, i], row_repeats)
                            for i in range(old_size[1])])[0]

    # perform column-wise interpolation on the columns of the matrix
    nrow, ncol = row_matrix.shape
    final_matrix = np.stack([np.repeat(row_matrix[i, :], col_repeats)
                             for i in range(nrow)])

    return final_matrix



def linear_interpolate(A, new_shape, res_A, new_res):
    x_A = np.arange(A.shape[1]) * res_A
    y_A = np.arange(A.shape[0]) * res_A
    x_A_new = np.arange(new_shape[1]) * new_res
    y_A_new = np.arange(new_shape[0]) * new_res
    A_interp = interp2d(x_A, y_A, np.squeeze(A))
    A_new = A_interp(x_A_new, y_A_new)

    return A_new
```

```
def geotiff_layer(img_filename, layer):

    """
    function to get individual layers of tiff files

    """

    # read in tiff file
    T = tiff.imread(img_filename)
    # access individual layer
    T = T[layer,:,:]

    return T



def interpolate_tifffile(img_id):

    """
    create data cube of 16 band data

    """

    pc_img_filename = 'data/geotiffs/{}_P.tif'.format(img_id)
    ms_img_filename = 'data/geotiffs/{}_M.tif'.format(img_id)
    swir_img_filename = 'data/geotiffs/{}_A.tif'.format(img_id)
    rgb_img_filename = 'data/geotiffs/{}.tif'.format(img_id)

    # read in panchromatic data
    pc_img = tiff.imread(pc_img_filename)

    # initialize empty list to create data cube
    cube = []
    # add panchromatic layer as first layer of data cube
    cube.append(pc_img)

    # add each multispectral layer to the data cube iteratively
    for ms_layer in range(8):
        # access each layer of the multispectral images
        ms_single_layer = geotiff_layer(ms_img_filename, ms_layer)
        # nearest-neighbor interpolate layer so that it is the same size
        # as the panchromatic data
        new_ms_layer = nn_interpolate(ms_single_layer, pc_img.shape)
        # add layer to the list
        cube.append(new_ms_layer)

    # add each SWIR layer to the data cube iteratively
```

```
    for swir_layer in range(8):
        # access each layer of the SWIR images
        swir_single_layer = geotiff_layer(swir_img_filename, swir_layer)
        # nearest-neighbor interpolate layer so that it is the same size as
        # the panchromatic data
        new_swir_layer = nn_interpolate(swir_single_layer, pc_img.shape)
        # add layer to the list
        cube.append(new_swir_layer)

    # add each RGB layer to the data cube iteratively
    for rgb_layer in range(3):
        # access each layer of the RGB images
        rgb_single_layer = geotiff_layer(rgb_img_filename, rgb_layer)
        # Do linear interpolation here because some times RGB array is bigger
        # than panchromatic array.
        # nn_interpolate doesn't work in such cases.
        if rgb_single_layer.shape != pc_img.shape:
            #rgb_single_layer = nn_interpolate(rgb_single_layer, pc_img.shape)
            rgb_single_layer = linear_interpolate(rgb_single_layer, pc_img.shape,
                                                  0.31, 0.31)
        # add layer to the list
        cube.append(rgb_single_layer)

    # return the list
    return cube
```

2.2 Labels Data

The preprocessing for the labels data, on the other hand, was different. Here, we had to convert the multipolygon data into binary masks before they can be tiled. We found the code posted by the Kaggle competition's third place finisher on Kaggle's website (https://www.kaggle.com/iglovikov/jaccard-polygons-polygons-mask-polygons) and borrowed it to move along our own data preprocessing. The code creating necessary functions for making the masks is displayed below.

```
In [ ]: """
        The code below is from the authors of the 3rd place submission for the
        Dstl Kaggle competition:
        https://www.kaggle.com/iglovikov/jaccard-polygons-polygons-mask-polygons

        """


        def _convert_coordinates_to_raster(coords, img_size, xymax):
            x_max, y_max = xymax
            height, width = img_size
            W1 = 1.0 * width * width / (width + 1)
            H1 = 1.0 * height * height / (height + 1)
```

```
    xf = W1 / x_max
    yf = H1 / y_max
    coords[:, 1] *= yf
    coords[:, 0] *= xf
    coords_int = np.round(coords).astype(np.int32)
    return coords_int



def _get_xmax_ymin(grid_sizes_panda, imageId):
    xmax,ymin=grid_sizes_panda[grid_sizes_panda.ImageId==imageId].iloc[0,1:].astype(floa
    return xmax, ymin



def _get_polygon_list(wkt_list_pandas, imageId, class_type):
    df_image = wkt_list_pandas[wkt_list_pandas.ImageId == imageId]
    multipoly_def = df_image[df_image.ClassType == class_type].MultipolygonWKT
    polygonList = None
    if len(multipoly_def) > 0:
        assert len(multipoly_def) == 1
        polygonList = wkt_loads(multipoly_def.values[0])
    return polygonList



def _get_and_convert_contours(polygonList, raster_img_size, xymax):
    perim_list = []
    interior_list = []
    if polygonList is None:
        return None
    for k in range(len(polygonList)):
        poly = polygonList[k]
        perim = np.array(list(poly.exterior.coords))
        perim_c = _convert_coordinates_to_raster(perim, raster_img_size,
                                                 xymax)
        perim_list.append(perim_c)
        for pi in poly.interiors:
            interior = np.array(list(pi.coords))
            interior_c = _convert_coordinates_to_raster(interior,
                                                        raster_img_size,
                                                        xymax)
            interior_list.append(interior_c)
    return perim_list, interior_list



def _plot_mask_from_contours(raster_img_size, contours, class_value=1):
```

```
    img_mask = np.zeros(raster_img_size, np.int8)
    if contours is None:
        return img_mask
    perim_list, interior_list = contours
    cv2.fillPoly(img_mask, perim_list, class_value)
    cv2.fillPoly(img_mask, interior_list, 0)
    return img_mask



def mask_to_polygons_layer(mask):
    all_polygons = []
    for shape, value in features.shapes(mask.astype(np.int16), mask=(mask == 1),
                                        transform=rasterio.Affine(1.0, 0, 0,
                                                                  0, 1.0, 0)):

        all_polygons.append(shapely.geometry.shape(shape))

    all_polygons = shapely.geometry.MultiPolygon(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 = shapely.geometry.MultiPolygon([all_polygons])
    return all_polygons



def generate_mask_for_image_and_class(raster_size, imageId, class_type,
                                      grid_sizes_panda, wkt_list_pandas):
    xymax = _get_xmax_ymin(grid_sizes_panda, imageId)
    polygon_list = _get_polygon_list(wkt_list_pandas, imageId, class_type)
    contours = _get_and_convert_contours(polygon_list, raster_size, xymax)
    mask = _plot_mask_from_contours(raster_size, contours, 1)
    return mask
```

In [6]:
```
def make_masks(img_id, shape, class_types, grid_sizes, train_wkt):

    """
    Make labels masks for an input image

    """

    num_classes = len(class_types)
    masks = []
    for ct in range(num_classes):
        mask = generate_mask_for_image_and_class(shape, img_id, ct+1,
```

```
                                                    grid_sizes, train_wkt)
            masks.append(mask.astype(bool))
        return masks


def plot_masks(masks):

    """
    Plot the masks

    """

    num_classes = len(masks)
    f, axs = plt.subplots(2, int(num_classes/2))
    f.set_size_inches(20,10)
    for ct in range(num_classes):
        r = int(np.floor(ct/5))
        c = ct - r*5
        axs[r,c].imshow(masks[ct])
        axs[r,c].set_title(class_types[ct])
        axs[r,c].set_yticklabels([])
        axs[r,c].set_xticklabels([])
```

2.3 Tiling the Data

The next step we took was to tile the data. Tiling the data allowed us to take the datacubes we had created from the satellite imagery data and the masks we creatd from the polygons and subset smaller cubes and squares, respectively, of a set size in order to train the model. We created a series of functions to randomly tile the datacubes and masks for a defined feature type. The functions can be found below

```
In [ ]: def tile_obs_rand(obs_cube, labels_list, class_type, tile_size, num_tiles):

            """
            Random tiling.
            Tile an observation and its labels into many smaller samples for a given class
            type,
            using random sampling

            """

            ts = int(tile_size/2)
            idx = CLASS_TYPES.index(class_type)
            data_tiles = []
            labels_tiles = []
            for t in range(num_tiles):
                rand_row = np.random.randint(ts + 1, obs_cube.shape[1]-(ts/2 + 1))
```

```
            rand_col = np.random.randint(ts + 1, obs_cube.shape[2]-(ts/2 + 1))
            data_tiles.append(obs_cube[:,rand_row-ts:rand_row+ts,rand_col-ts:rand_col+ts])
            labels_tiles.append(labels_list[idx][rand_row-ts:rand_row+ts,rand_col-ts:rand_co

        return data_tiles, labels_tiles



def blockshaped(arr, nrows, ncols):

    """
    Break input array into blocks, each of shape (nrows, ncols)

    Return an array of shape (n, nrows, ncols) where n * nrows * ncols = arr.size

    """

    h, w = arr.shape
    return (arr.reshape(h//nrows, nrows, -1, ncols)
               .swapaxes(1,2)
               .reshape(-1, nrows, ncols))



def tile_obs(obs_cube, labels_list, class_type, tile_size):

    """
    Checker board tiling
    Uses "blockshaped" function

    """

    # Crop area for tiling
    wx = obs_cube.shape[2]
    wy = obs_cube.shape[1]
    ntilex = wx//tile_size
    ntiley = wy//tile_size
    buff_left = int(np.remainder(wx, ntilex*tile_size)/2)
    buff_right = np.remainder(wx, ntilex*tile_size) - buff_left
    buff_up = int(np.remainder(wy, ntiley*tile_size)/2)
    buff_down = np.remainder(wy, ntiley*tile_size) - buff_up
    obs_cube = obs_cube[:, buff_up:wy-buff_down, buff_left:wx-buff_right]
    idx = CLASS_TYPES.index(class_type)
    labels_crop = labels_list[idx][buff_up:wy-buff_down, buff_left:wx-buff_right]

    # Tile spectral data
    layers_list = []
    num_layers = obs_cube.shape[0]
```

```
    for layer in range(num_layers):
        layers_list.append(blockshaped(obs_cube[layer,:,:], tile_size, tile_size))

    num_tiles = layers_list[0].shape[0]
    data_tiles = []
    for t in range(num_tiles):
        tile = np.zeros((num_layers, tile_size, tile_size))
        for layer in range(num_layers):
            tile[layer, :, :] = layers_list[layer][t, :, :]
        data_tiles.append(tile.astype('uint16'))

    # Tile labels data
    labels_tiles = blockshaped(labels_crop, tile_size, tile_size)
    labels_tiles=[labels_tiles[r,:,:].astype('uint16') for r in range(labels_tiles.shape

    return data_tiles, labels_tiles
```

2.4 Creating the Data

The final step we took in the preprocessing involved combining all of the aforementioned steps into one function which could automatically run the preprocessing. This funtion took inputs in the form of a feature type and a tile size and saved the resulting image and mask tiles in a data folder. These image and mask tiles could then be used in the next section to train our model. The code creating the function which made the training data is below.

```
In [ ]: def make_train_data(feature, tile_size):

            """
            1. Choose an image
            2. Load and interpolate spectral data
            3. Load wkt data and generate mask for feature of interest
            4. Tile spectral data and mask
            5. Save data and masks

            """

            # Only look at images with at least the minimum number of classes
            img_ids=train_images.loc[train_images['ClassCount']>=MIN_CLASSES]['ImageId']

            data_tiles = []
            labels_tiles = []
            for ID in img_ids:
                print('Preparing', ID)
                # Spectral data
                interp_cube = interpolate_tifffile(ID)
                data_cube = np.stack(interp_cube, axis=0)
                shape = data_cube.shape[1:]
                # Labels data
```

```
            gs = pd.read_csv('data/grid_sizes.csv',
                             names=['ImageId', 'Xmax', 'Ymin'],
                             skiprows=1)
            train_wkt = pd.read_csv('data/train_wkt_v4.csv')
            feature_masks = make_masks(ID, shape, CLASS_TYPES, gs, train_wkt)
            # Tiling
            data_tiles_temp,labels_tiles_temp = tile_obs(data_cube,
                                                         feature_masks,
                                                         feature,
                                                         tile_size)
            # Concatenate
            data_tiles = data_tiles + data_tiles_temp
            labels_tiles = labels_tiles + labels_tiles_temp

        # Save
        if not os.path.isdir(DATA_DIR):
            os.mkdir(DATA_DIR);
        # Images
        fn_images = DATA_DIR + '/images'
        np.save(fn_images, data_tiles)
        # Masks
        fn_masks = DATA_DIR + '/masks'
        np.save(fn_masks, labels_tiles)

        # Print statement
        print('Made', len(data_tiles),
              'image tiles with shape', data_tiles[0].shape)
        print('Made', len(labels_tiles),
              'mask tiles with shape', labels_tiles[0].shape)

        return data_tiles, labels_tiles
```

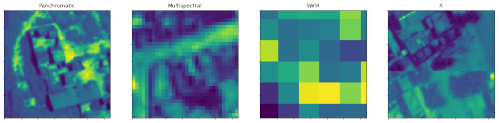
```
In [ ]: images, masks = make_train_data(FEATURE, TILE_SIZE)
```

The following two code chunks display a few examples of the tiled data for both the imagery and the masks that will be fed into the U-net.

```
In [10]: f, (ax1, ax2, ax3, ax4) = plt.subplots(1,4)
         f.set_size_inches(20,10);

         ax1.imshow(images[10][0,:,:]);
         ax1.set_yticklabels([]);
         ax1.set_xticklabels([]);
         ax1.set_title('Panchromatic')
         ax2.imshow(images[20][5,:,:]);
         ax2.set_yticklabels([]);
         ax2.set_xticklabels([]);
         ax2.set_title('Multispectral')
```

```
ax3.imshow(images[40][10,:,:]);
ax3.set_yticklabels([]);
ax3.set_xticklabels([]);
ax3.set_title('SWIR')
ax4.imshow(images[50][17,:,:]);
ax4.set_yticklabels([]);
ax4.set_xticklabels([]);
ax4.set_title('R')

print('Some examples of imagery at different wave bands and locations')
plt.show()
```

```
Some examples of imagery at different wave bands and locations
```



In [11]:
```
f, (ax1, ax2, ax3, ax4) = plt.subplots(1,4)
f.set_size_inches(20,10);

ax1.imshow(masks[10]);
ax1.set_yticklabels([]);
ax1.set_xticklabels([]);
ax2.imshow(masks[20]);
ax2.set_yticklabels([]);
ax2.set_xticklabels([]);
ax3.imshow(masks[40]);
ax3.set_yticklabels([]);
ax3.set_xticklabels([]);
ax4.imshow(masks[50]);
ax4.set_yticklabels([]);
ax4.set_xticklabels([]);

print('Feature masks corresponding to above locations')
plt.show()
```

```
Feature masks corresponding to above locations
```

3 Models

```
In [ ]: import numpy as np
        import os
        import skimage.io as io
        import skimage.transform as trans
        import numpy as np
        import keras
        from keras.models import *
        from keras.layers import *
        from keras.optimizers import *
        from keras.callbacks import ModelCheckpoint, LearningRateScheduler
        from keras import backend as K
        from keras.backend import binary_crossentropy
        from keras.preprocessing.image import ImageDataGenerator
        import matplotlib.pyplot as plt
```

```
In [ ]: K.tensorflow_backend._get_available_gpus()
```

3.1 Global parameters

We used a batch size of 128, with 200 steps per epoch for 5 epochs. The train/test split was 0.8. The `FEATURE_PRESENCE_MIN` variable is the (weak) minimum feature presence we require in an image to consider it in our training, measured as a percentage of all pixels in the image. A `FEATURE_PRESENCE_MIN` of 0 corresponds to looking at all training images. We wanted to create this variable because we wanted to test our models on how well they found features that exist in the images, and rare classes would have stopped us from achieving this goal. We ran tests with 0 `FEATURE_PRESENCE_MIN` and 0.04 `FEATURE_PRESENCE_MIN`, which means 4% of pixels in a mask should correspond to features.

```
In [ ]: BATCH_SIZE = 128
        NUM_STEPS = 200
        NUM_EPOCHS = 5
        TRAIN_TEST_SPLIT = 0.8
        FEATURE_PRESENCE_MIN = 0.04
```

3.2 Load data

The input images are 128x128 pixels, with 20 channels each, as explained previously. The output masks are binary 128x128x1 arrays with 0 where the feature is not present and 1 where the feature is present in the corresponding image.

```
In [ ]: # Load
        images = np.load('data/data_tiles/images.npy')
        masks = np.load('data/data_tiles/masks_road.npy')

        # Convert to numpy arrays
        images = np.array(images)
        images = np.swapaxes(images, 1, 2)
```

```
        images = np.swapaxes(images, 2, 3)
        masks = np.array(masks).reshape(-1,128,128,1)

        # Print statements
        print('Image array has shape', images.shape)
        print('Mask array has shape', masks.shape)
```

```
In [21]: # Only look at data with the class present
         idxs=(np.mean(masks.reshape(masks.shape[0], -1), axis=1)>=FEATURE_PRESENCE_MIN)
         masks = masks[idxs]
         images = images[idxs]

         # Print statements
         print('Filtered image array has shape', images.shape)
         print('Filtered mask array has shape', masks.shape)
```

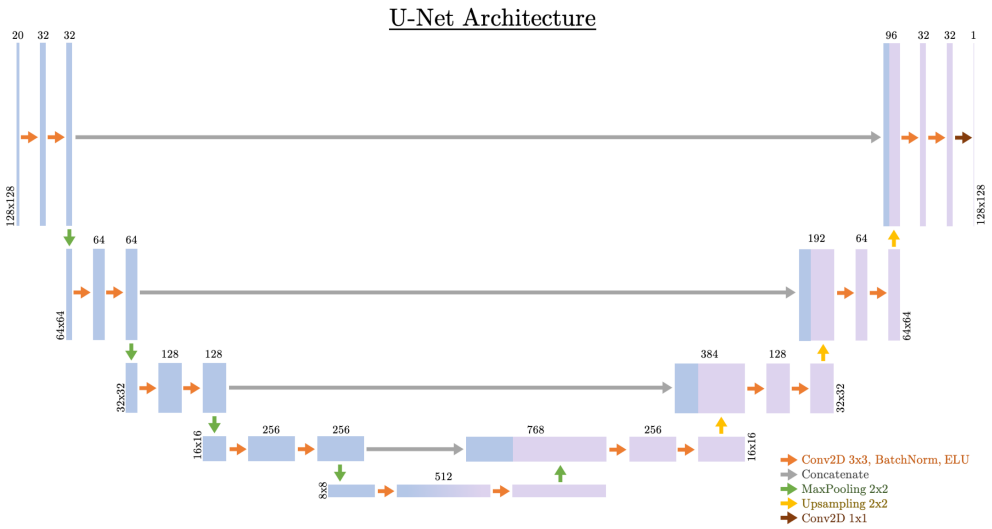
```
Filtered image array has shape (139, 128, 128, 20)
Filtered mask array has shape (139, 128, 128, 1)
```

```
In [22]: # Train-test split
         train_idxs = np.random.choice(np.arange(0,images.shape[0]),
                                       int(TRAIN_TEST_SPLIT*images.shape[0]),
                                       replace=False)
         index_msk = np.zeros(images.shape[0])
         index_msk[train_idxs] = 1
         X_train = images[train_idxs]
         y_train = masks[train_idxs]
         X_test = images[~index_msk.astype(bool)]
         y_test = masks[~index_msk.astype(bool)]

         # Print statements
         print('X_train has shape', X_train.shape)
         print('y_train has shape', y_train.shape)
         print('X_test has shape', X_test.shape)
         print('y_test has shape', y_test.shape)
```

```
X_train has shape (111, 128, 128, 20)
y_train has shape (111, 128, 128, 1)
X_test has shape (28, 128, 128, 20)
y_test has shape (28, 128, 128, 1)
```

3.3 Model Architecture

For our model, we used a U-net architecture:



U-nets are fully convolutional networks that are similar to autoencoders in that they encode the image down into a latent space, then decode the latent space to reconstruct a target image. However, autoencoders use the input image as the target, with the goal being a good reconstruction of the input image from the latent space. With U-nets, we want to reconstruct the segmentation mask that corresponds to where the features are present. So, U-nets introduce "skip connections" into the network, which concatenate feature representations in the encoder with their corresponding feature representations in the decoder. These skip connections allow the U-net to simultaneously learn abstract features from the latent space and reintroduce the features captured by the encoder to maximize the amount of information the network sees. Ronneberger et al. show that these connections create a better segmentation result, probably because the network sees the low-abstraction features in the original image, which should generate a better reconstruction than relying solely on the latent space. For autoencoders, these skip connections would give unfairly give the model an easier reconstruction, which is why they are not used.

We chose this network based on the 3rd place Kaggle submission (Iglovikov et al.), and our architecture is very similar. In one of their models, Iglovikov et al. use dropout layers as well, but we removed those layers from the model to overfit the network more to the masks, as our goal is to reconstruct the masks perfectly. We also tested the mdoel with Dropout. Our final model had 7,863,329 parameters.

This section of our report contains a detailed look at how our model runs for one specific input. We then created a function where we could easily test hyperparameters, run the model, visualize the results, and evaluate the performance, all in the following section.

```
In [23]: img_rows = 128
         img_cols = 128
         smooth = 1e-12
         num_channels = 20
         num_mask_channels = 1

         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_unet():
    inputs = Input((img_rows, img_cols, num_channels))
    conv1 = Conv2D(32, 3, padding='same', kernel_initializer='he_uniform')(inputs)
    conv1 = BatchNormalization()(conv1)
    conv1 = keras.layers.advanced_activations.ELU()(conv1)
    conv1 = Conv2D(32, 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, padding='same', kernel_initializer='he_uniform')(pool1)
    conv2 = BatchNormalization()(conv2)
    conv2 = keras.layers.advanced_activations.ELU()(conv2)
    conv2 = Conv2D(64, 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, padding='same', kernel_initializer='he_uniform')(pool2)
    conv3 = BatchNormalization()(conv3)
    conv3 = keras.layers.advanced_activations.ELU()(conv3)
    conv3 = Conv2D(128, 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, padding='same', kernel_initializer='he_uniform')(pool3)
    conv4 = BatchNormalization()(conv4)
```

```
conv4 = keras.layers.advanced_activations.ELU()(conv4)
conv4 = Conv2D(256, 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, padding='same', kernel_initializer='he_uniform')(pool4)
conv5 = BatchNormalization()(conv5)
conv5 = keras.layers.advanced_activations.ELU()(conv5)
conv5 = Conv2D(512, 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])
conv6 = Conv2D(256, 3, padding='same', kernel_initializer='he_uniform')(up6)
conv6 = BatchNormalization()(conv6)
conv6 = keras.layers.advanced_activations.ELU()(conv6)
conv6 = Conv2D(256, 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])
conv7 = Conv2D(128, 3, padding='same', kernel_initializer='he_uniform')(up7)
conv7 = BatchNormalization()(conv7)
conv7 = keras.layers.advanced_activations.ELU()(conv7)
conv7 = Conv2D(128, 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])
conv8 = Conv2D(64, 3, padding='same', kernel_initializer='he_uniform')(up8)
conv8 = BatchNormalization()(conv8)
conv8 = keras.layers.advanced_activations.ELU()(conv8)
conv8 = Conv2D(64, 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])
conv9 = Conv2D(32, 3, padding='same', kernel_initializer='he_uniform')(up9)
conv9 = BatchNormalization()(conv9)
conv9 = keras.layers.advanced_activations.ELU()(conv9)
conv9 = Conv2D(32, 3, padding='same', kernel_initializer='he_uniform')(conv9)
conv9 = BatchNormalization()(conv9)
conv9 = keras.layers.advanced_activations.ELU()(conv9)
conv10 = Conv2D(num_mask_channels, 1, activation='sigmoid')(conv9)

model = Model(inputs=inputs, outputs=conv10)
```

```
    return model
```

In [24]:
```
model = get_unet()
model.summary()
```

```
_________________________________________________________________________________________________
Layer (type)                 Output Shape          Param #     Connected to
=================================================================================================
input_2 (InputLayer)         (None, 128, 128, 20) 0
_________________________________________________________________________________________________
conv2d_20 (Conv2D)           (None, 128, 128, 32) 5792        input_2[0][0]
_________________________________________________________________________________________________
batch_normalization_19 (BatchNo (None, 128, 128, 32) 128         conv2d_20[0][0]
_________________________________________________________________________________________________
elu_19 (ELU)                 (None, 128, 128, 32) 0           batch_normalization_19[0][0]
_________________________________________________________________________________________________
conv2d_21 (Conv2D)           (None, 128, 128, 32) 9248        elu_19[0][0]
_________________________________________________________________________________________________
batch_normalization_20 (BatchNo (None, 128, 128, 32) 128         conv2d_21[0][0]
_________________________________________________________________________________________________
elu_20 (ELU)                 (None, 128, 128, 32) 0           batch_normalization_20[0][0]
_________________________________________________________________________________________________
max_pooling2d_5 (MaxPooling2D)  (None, 64, 64, 32)   0           elu_20[0][0]
_________________________________________________________________________________________________
conv2d_22 (Conv2D)           (None, 64, 64, 64)   18496       max_pooling2d_5[0][0]
_________________________________________________________________________________________________
batch_normalization_21 (BatchNo (None, 64, 64, 64)   256         conv2d_22[0][0]
_________________________________________________________________________________________________
elu_21 (ELU)                 (None, 64, 64, 64)   0           batch_normalization_21[0][0]
_________________________________________________________________________________________________
conv2d_23 (Conv2D)           (None, 64, 64, 64)   36928       elu_21[0][0]
_________________________________________________________________________________________________
batch_normalization_22 (BatchNo (None, 64, 64, 64)   256         conv2d_23[0][0]
_________________________________________________________________________________________________
elu_22 (ELU)                 (None, 64, 64, 64)   0           batch_normalization_22[0][0]
_________________________________________________________________________________________________
max_pooling2d_6 (MaxPooling2D)  (None, 32, 32, 64)   0           elu_22[0][0]
_________________________________________________________________________________________________
conv2d_24 (Conv2D)           (None, 32, 32, 128)  73856       max_pooling2d_6[0][0]
_________________________________________________________________________________________________
batch_normalization_23 (BatchNo (None, 32, 32, 128)  512         conv2d_24[0][0]
_________________________________________________________________________________________________
elu_23 (ELU)                 (None, 32, 32, 128)  0           batch_normalization_23[0][0]
_________________________________________________________________________________________________
conv2d_25 (Conv2D)           (None, 32, 32, 128)  147584      elu_23[0][0]
_________________________________________________________________________________________________
batch_normalization_24 (BatchNo (None, 32, 32, 128)  512         conv2d_25[0][0]
_________________________________________________________________________________________________
```

```
elu_24 (ELU)                    (None, 32, 32, 128)  0          batch_normalization_24[0][0]
__________________________________________________________________________________________________
max_pooling2d_7 (MaxPooling2D)  (None, 16, 16, 128)  0          elu_24[0][0]
__________________________________________________________________________________________________
conv2d_26 (Conv2D)              (None, 16, 16, 256)  295168     max_pooling2d_7[0][0]
__________________________________________________________________________________________________
batch_normalization_25 (BatchNo (None, 16, 16, 256)  1024       conv2d_26[0][0]
__________________________________________________________________________________________________
elu_25 (ELU)                    (None, 16, 16, 256)  0          batch_normalization_25[0][0]
__________________________________________________________________________________________________
conv2d_27 (Conv2D)              (None, 16, 16, 256)  590080     elu_25[0][0]
__________________________________________________________________________________________________
batch_normalization_26 (BatchNo (None, 16, 16, 256)  1024       conv2d_27[0][0]
__________________________________________________________________________________________________
elu_26 (ELU)                    (None, 16, 16, 256)  0          batch_normalization_26[0][0]
__________________________________________________________________________________________________
max_pooling2d_8 (MaxPooling2D)  (None, 8, 8, 256)    0          elu_26[0][0]
__________________________________________________________________________________________________
conv2d_28 (Conv2D)              (None, 8, 8, 512)    1180160    max_pooling2d_8[0][0]
__________________________________________________________________________________________________
batch_normalization_27 (BatchNo (None, 8, 8, 512)    2048       conv2d_28[0][0]
__________________________________________________________________________________________________
elu_27 (ELU)                    (None, 8, 8, 512)    0          batch_normalization_27[0][0]
__________________________________________________________________________________________________
conv2d_29 (Conv2D)              (None, 8, 8, 512)    2359808    elu_27[0][0]
__________________________________________________________________________________________________
batch_normalization_28 (BatchNo (None, 8, 8, 512)    2048       conv2d_29[0][0]
__________________________________________________________________________________________________
elu_28 (ELU)                    (None, 8, 8, 512)    0          batch_normalization_28[0][0]
__________________________________________________________________________________________________
up_sampling2d_5 (UpSampling2D)  (None, 16, 16, 512)  0          elu_28[0][0]
__________________________________________________________________________________________________
concatenate_5 (Concatenate)     (None, 16, 16, 768)  0          up_sampling2d_5[0][0]
                                                                elu_26[0][0]
__________________________________________________________________________________________________
conv2d_30 (Conv2D)              (None, 16, 16, 256)  1769728    concatenate_5[0][0]
__________________________________________________________________________________________________
batch_normalization_29 (BatchNo (None, 16, 16, 256)  1024       conv2d_30[0][0]
__________________________________________________________________________________________________
elu_29 (ELU)                    (None, 16, 16, 256)  0          batch_normalization_29[0][0]
__________________________________________________________________________________________________
conv2d_31 (Conv2D)              (None, 16, 16, 256)  590080     elu_29[0][0]
__________________________________________________________________________________________________
batch_normalization_30 (BatchNo (None, 16, 16, 256)  1024       conv2d_31[0][0]
__________________________________________________________________________________________________
elu_30 (ELU)                    (None, 16, 16, 256)  0          batch_normalization_30[0][0]
__________________________________________________________________________________________________
up_sampling2d_6 (UpSampling2D)  (None, 32, 32, 256)  0          elu_30[0][0]
```

```
concatenate_6 (Concatenate)     (None, 32, 32, 384)  0           up_sampling2d_6[0][0]
                                                                 elu_24[0][0]
conv2d_32 (Conv2D)              (None, 32, 32, 128)  442496      concatenate_6[0][0]
batch_normalization_31 (BatchNo (None, 32, 32, 128)  512         conv2d_32[0][0]
elu_31 (ELU)                    (None, 32, 32, 128)  0           batch_normalization_31[0][0]
conv2d_33 (Conv2D)              (None, 32, 32, 128)  147584      elu_31[0][0]
batch_normalization_32 (BatchNo (None, 32, 32, 128)  512         conv2d_33[0][0]
elu_32 (ELU)                    (None, 32, 32, 128)  0           batch_normalization_32[0][0]
up_sampling2d_7 (UpSampling2D)  (None, 64, 64, 128)  0           elu_32[0][0]
concatenate_7 (Concatenate)     (None, 64, 64, 192)  0           up_sampling2d_7[0][0]
                                                                 elu_22[0][0]
conv2d_34 (Conv2D)              (None, 64, 64, 64)   110656      concatenate_7[0][0]
batch_normalization_33 (BatchNo (None, 64, 64, 64)   256         conv2d_34[0][0]
elu_33 (ELU)                    (None, 64, 64, 64)   0           batch_normalization_33[0][0]
conv2d_35 (Conv2D)              (None, 64, 64, 64)   36928       elu_33[0][0]
batch_normalization_34 (BatchNo (None, 64, 64, 64)   256         conv2d_35[0][0]
elu_34 (ELU)                    (None, 64, 64, 64)   0           batch_normalization_34[0][0]
up_sampling2d_8 (UpSampling2D)  (None, 128, 128, 64) 0           elu_34[0][0]
concatenate_8 (Concatenate)     (None, 128, 128, 96) 0           up_sampling2d_8[0][0]
                                                                 elu_20[0][0]
conv2d_36 (Conv2D)              (None, 128, 128, 32) 27680       concatenate_8[0][0]
batch_normalization_35 (BatchNo (None, 128, 128, 32) 128         conv2d_36[0][0]
elu_35 (ELU)                    (None, 128, 128, 32) 0           batch_normalization_35[0][0]
conv2d_37 (Conv2D)              (None, 128, 128, 32) 9248        elu_35[0][0]
batch_normalization_36 (BatchNo (None, 128, 128, 32) 128         conv2d_37[0][0]
```

```
elu_36 (ELU)                    (None, 128, 128, 32) 0           batch_normalization_36[0][0]
__________________________________________________________________________________________________
conv2d_38 (Conv2D)              (None, 128, 128, 1)  33          elu_36[0][0]
==================================================================================================
Total params: 7,863,329
Trainable params: 7,857,441
Non-trainable params: 5,888
__________________________________________________________________________________________________
```

```
In [25]: # Compile model
         model.compile(optimizer=Nadam(lr=1e-3),
                       loss=jaccard_coef_loss,
                       metrics=['binary_crossentropy', jaccard_coef_int])
```

For our loss function, we used a combination of binary cross-entropy and the Jaccard index. The Jaccard index, also called Intersection over Union, is a measure of similarity between two sets, and is defined as follows:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

The DSTL satellite Kaggle competition used the Jaccard score to evaluate entrants' submissions, so we wanted to include it in our loss function. We used Iglovikov et al. loss function:

$$\mathcal{L}(M_{tr}, M_{te}) = -\log\left(J(M_{tr}, M_{te})\right) + \mathcal{L}_{\text{Bin CE}}(M_{tr}, M_{te})$$

where M_{tr}, M_{te} are the training and testing masks, respectively. This loss function takes the negative log of the Jaccard score, and adds the binary cross-entropy loss. We also tested just using the negative log of the Jaccard score and binary cross-entropy loss functions separately as well.

3.4 Model Training and Data Augmentation

We trained our model using two ImageDataGenerators with augmentation. We standardized the data to mean 0 and variance 1, and introduced random rotations, width- and height-shifts, zooms, and horizontal flips via Keras's ImageDataGenerator class. We synchronized the random augmentations via the same random seed, and found that the masks and images were synchronized by visual inspection (output below).

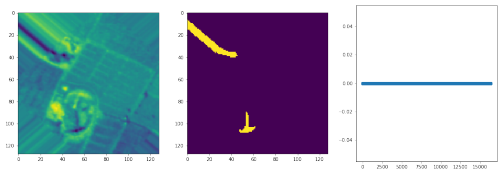
```
In [26]: def preprofunc(im):
             im = (im > 0.5)
             return im
```

```
In [27]: nn = X_train.shape[0]
         train_nn = int(nn * 0.9)
         X_train = X_train[0:train_nn,:,:,:]
         X_val = X_train[train_nn:,:,:,:]
         y_train = y_train[0:train_nn,:,:,:]
         y_val = y_train[train_nn:,:,:,:]

         # We create two instances with the same arguments
```

```
        image_datagen = ImageDataGenerator(featurewise_center=True,
                             featurewise_std_normalization=True,
                             rotation_range=90,
                             width_shift_range=0.2,
                             height_shift_range=0.2,
                             zoom_range=0.2,
                             horizontal_flip = True)
        mask_datagen = ImageDataGenerator(rotation_range=90,
                             width_shift_range=0.2,
                             height_shift_range=0.2,
                             zoom_range=0.2,
                             horizontal_flip = True,
                             preprocessing_function = preprofunc)

        # Provide the same seed and keyword arguments to the fit and flow methods
        seed = 1
        image_datagen.fit(X_train, augment=True, seed=seed)
        mask_datagen.fit(y_train, augment=True, seed=seed)

        image_generator = image_datagen.flow(
                             X_train,
                             seed=seed,
                             batch_size=BATCH_SIZE)
        mask_generator = mask_datagen.flow(
                             y_train,
                             seed=seed,
                             batch_size=BATCH_SIZE)

        image_generator_val = image_datagen.flow(
                                 X_val,
                                 seed=seed,
                                 batch_size=BATCH_SIZE)
        mask_generator_val = mask_datagen.flow(
                                 y_val,
                                 seed=seed,
                                 batch_size=BATCH_SIZE)

        # Combine generators into one which yields image and masks
        train_generator = zip(image_generator, mask_generator)
        validation_generator = zip(image_generator_val, mask_generator_val)
```

```
/usr/share/anaconda3/lib/python3.6/site-packages/keras_preprocessing/image.py:1358: UserWarning:
  ' channels).')
/usr/share/anaconda3/lib/python3.6/site-packages/keras_preprocessing/image.py:1643: UserWarning:
  str(self.x.shape[channels_axis]) + ' channels).')
/usr/share/anaconda3/lib/python3.6/site-packages/keras_preprocessing/image.py:1643: UserWarning:
  str(self.x.shape[channels_axis]) + ' channels).')
```

```
In [28]: # Make sure generator is working properly
         x = image_generator[0]
         y = mask_generator[0]

         idx = 1

         plt.rcParams['figure.figsize'] = (18, 6);
         f, (ax1, ax2, ax3) = plt.subplots(1,3)
         ax1.imshow(x[idx][:,:,0]);
         ax2.imshow(y[idx][:,:,0]);
         ax3.plot(y[2][:,:,0].ravel(),'o');
```



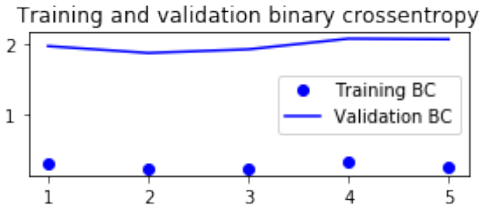
Above, the augmented images and masks line up together well.
Then, we trained our model using the zipped train generator.

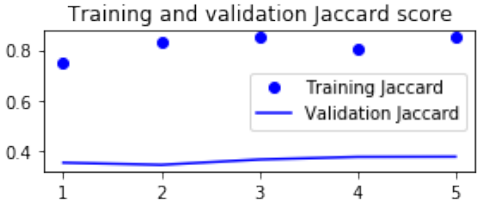
```
In [29]: history = model.fit_generator(train_generator,
                                       steps_per_epoch=NUM_STEPS,
                                       epochs=NUM_EPOCHS,
                                       validation_data=validation_generator,
                                       validation_steps=10)
```

```
Epoch 1/5
400/400 [==============================] - 1120s 3s/step - loss: 1.3629 - binary_crossentropy: 0
Epoch 2/5
400/400 [==============================] - 1103s 3s/step - loss: 0.7647 - binary_crossentropy: 0
Epoch 3/5
400/400 [==============================] - 1129s 3s/step - loss: 0.6473 - binary_crossentropy: 0
Epoch 4/5
400/400 [==============================] - 1145s 3s/step - loss: 0.8891 - binary_crossentropy: 0
Epoch 5/5
400/400 [==============================] - 1114s 3s/step - loss: 0.6495 - binary_crossentropy: 0
```

```
In [34]: bc = history.history['binary_crossentropy']
         val_bc = history.history['val_binary_crossentropy']
         jac = history.history['jaccard_coef_int']
         val_jac = history.history['val_jaccard_coef_int']
         loss = history.history['loss']
         val_loss = history.history['val_loss']

         epochs = range(1, len(bc) + 1)
```

```
plt.rcParams['figure.figsize'] = (4.5, 1.5);

plt.plot(epochs, bc, 'bo', label='Training BC')
plt.plot(epochs, val_bc, 'b', label='Validation BC')
plt.title('Training and validation binary crossentropy')
plt.legend()

plt.figure()

plt.plot(epochs, jac, 'bo', label='Training Jaccard')
plt.plot(epochs, val_jac, 'b', label='Validation Jaccard')
plt.title('Training and validation Jaccard score')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```






```
In [35]: model.save('model_standingWater_someTiles_5epoch.hdf5')
```

```
In [76]: def evaluate_model():
             image_train_datagen = ImageDataGenerator(featurewise_center=True,
                                   featurewise_std_normalization=True)

             image_train_datagen.fit(X_train, augment=True, seed=seed)

             image_train_generator = image_train_datagen.flow(
                                 X_train, y_train,
                                 seed=seed,
                                 batch_size=X_train.shape[0])

             loss, bin_cross_entropy, jaccard = model.evaluate(image_train_generator[0][0],
                                                               image_train_generator[0][1],
                                                               verbose=0)
             print("Train Loss:", loss)
             print("Train Binary Cross Entropy:", bin_cross_entropy)
             print("Train Jaccard Score:", jaccard)

             print()
             image_test_datagen = ImageDataGenerator(featurewise_center=True,
                                  featurewise_std_normalization=True)

             image_test_datagen.fit(X_test, augment=True, seed=seed)

             image_test_generator = image_test_datagen.flow(
                                 X_test, y_test,
                                 seed=seed,
                                 batch_size=X_test.shape[0])

             loss, bin_cross_entropy, jaccard = model.evaluate(image_test_generator[0][0],
                                                               image_test_generator[0][1],
                                                               verbose=0)
             print("Test Loss:", loss)
             print("Test Binary Cross Entropy:", bin_cross_entropy)
             print("Test Jaccard Score:", jaccard)
```

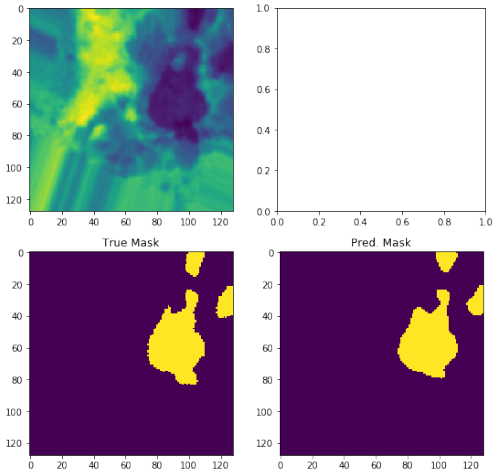
```
In [77]: evaluate_model()
```

```
Train Loss: 1.01386736642133
Train Binary Cross Entropy: 0.5212819984367302
Train Jaccard Score: 0.7786112207550187

Test Loss: 1.414794921875
Test Binary Cross Entropy: 0.6278472542762756
Test Jaccard Score: 0.7163583636283875
```

3.5 Prediction

To check our results, we wrote this function that allowed us to visualize layers of images with their true and predicted feature masks.

```
In [36]: X_example_train = image_generator[0]
         y_example_train = mask_generator[0]
         y_example_pred_train = model.predict(X_example_train)
         X_example_test = image_generator_val[0]
         y_example_test = mask_generator_val[0]
         y_example_pred_test = model.predict(X_example_test)

In [37]: def check_results(i, train=True):

             if train:
                 X_example = X_example_train
                 y_example = y_example_train
                 y_example_pred = y_example_pred_train
             else:
                 X_example = X_example_test
                 y_example = y_example_test
                 y_example_pred = y_example_pred_test

             fig, axs = plt.subplots(2,2, figsize=(9,9))
             axs = axs.flat
             axs[0].imshow(X_example[i,:,:,18])
             axs[2].imshow(y_example[i,:,:,0]); axs[2].set_title("True Mask")
             axs[3].imshow((y_example_pred[i,:,:,0] > .5)); axs[3].set_title("Pred. Mask")
             plt.show()

In [39]: check_results(2, train=False)
```



Based on the above visualization, the predicted mask has a very good fit to the true mask. ## Automatic Running We didn't want to have to rerun our notebook every time we wanted to test hyperparameters, so we designed an easy function that builds, trains, visualizes, saves, and evaluates a model given some hyperparameters.

```
In [4]: def do_model(feature, channels, FEATURE_PRESENCE_MIN=FEATURE_PRESENCE_MIN,
                    lr_decay = None, loss = None):
            print('Starting...')
            # Load
            mask_str = 'data/data_tiles/masks_' + feature + '.npy'
            masks = np.load(mask_str)
            if channels == 1:
                images = np.load('data/data_tiles/images_panchromatic.npy')
                images = np.array(images)
                images.reshape(-1, 128, 128,1)
            elif channels == 3:
                images = np.load('data/data_tiles/images_3dim.npy')
                images = np.array(images)
                # Convert to numpy arrays
                images = np.swapaxes(images, 1, 2)
                images = np.swapaxes(images, 2, 3)
                masks = np.array(masks).reshape(-1,128,128,1)
            elif channels == 20:
                images = np.load('data/data_tiles/images.npy')
                images = np.array(images)
                images = np.swapaxes(images, 1, 2)
                images = np.swapaxes(images, 2, 3)
                masks = np.array(masks).reshape(-1,128,128,1)
            else:
                print("Error, incorrect number of channels")
                return 0
```

```
if lr_decay is None:
    lr_decay = 0.004


# Only look at data with the class present
idxs = (np.mean(masks.reshape(masks.shape[0], -1), axis=1)>=FEATURE_PRESENCE_MIN)
masks = masks[idxs]
images = images[idxs]

# Train-test split
train_idxs = np.random.choice(np.arange(0,images.shape[0]),
                              int(TRAIN_TEST_SPLIT*images.shape[0]),
                              replace=False)
index_msk = np.zeros(images.shape[0])
index_msk[train_idxs] = 1
X_train = images[train_idxs]
y_train = masks[train_idxs]
X_test = images[~index_msk.astype(bool)]
y_test = masks[~index_msk.astype(bool)]


print("Done data processing...")

# # # # # # #
# MODEL
# # # # # # #
img_rows = 128
img_cols = 128
smooth = 1e-12
num_channels = channels
num_mask_channels = 1

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 just_jaccard_coef_loss(y_true, y_pred):
    return -K.log(jaccard_coef(y_true, y_pred))

def bin_ce(y_true, y_pred):
    return binary_crossentropy(y_pred, y_true)

if loss == 'binary_crossentropy':
    jaccard_coef_loss = bin_ce
elif loss == 'jaccard':
    jaccard_coef_loss = just_jaccard_coef_loss


def get_unet():
    inputs = Input((img_rows, img_cols, num_channels))
    conv1 = Conv2D(32, 3, padding='same', kernel_initializer='he_uniform')(inputs)
    conv1 = BatchNormalization()(conv1)
    conv1 = keras.layers.advanced_activations.ELU()(conv1)
    conv1 = Conv2D(32, 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, padding='same', kernel_initializer='he_uniform')(pool1)
    conv2 = BatchNormalization()(conv2)
    conv2 = keras.layers.advanced_activations.ELU()(conv2)
    conv2 = Conv2D(64, 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, padding='same', kernel_initializer='he_uniform')(pool2)
    conv3 = BatchNormalization()(conv3)
    conv3 = keras.layers.advanced_activations.ELU()(conv3)
    conv3 = Conv2D(128, 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, padding='same', kernel_initializer='he_uniform')(pool3)
    conv4 = BatchNormalization()(conv4)
```

```
conv4 = keras.layers.advanced_activations.ELU()(conv4)
conv4 = Conv2D(256, 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, padding='same', kernel_initializer='he_uniform')(pool4)
conv5 = BatchNormalization()(conv5)
conv5 = keras.layers.advanced_activations.ELU()(conv5)
conv5 = Conv2D(512, 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])
conv6 = Conv2D(256, 3, padding='same', kernel_initializer='he_uniform')(up6)
conv6 = BatchNormalization()(conv6)
conv6 = keras.layers.advanced_activations.ELU()(conv6)
conv6 = Conv2D(256, 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])
conv7 = Conv2D(128, 3, padding='same', kernel_initializer='he_uniform')(up7)
conv7 = BatchNormalization()(conv7)
conv7 = keras.layers.advanced_activations.ELU()(conv7)
conv7 = Conv2D(128, 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])
conv8 = Conv2D(64, 3, padding='same', kernel_initializer='he_uniform')(up8)
conv8 = BatchNormalization()(conv8)
conv8 = keras.layers.advanced_activations.ELU()(conv8)
conv8 = Conv2D(64, 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])
conv9 = Conv2D(32, 3, padding='same', kernel_initializer='he_uniform')(up9)
conv9 = BatchNormalization()(conv9)
conv9 = keras.layers.advanced_activations.ELU()(conv9)
conv9 = Conv2D(32, 3, padding='same', kernel_initializer='he_uniform')(conv9)
conv9 = BatchNormalization()(conv9)
conv9 = keras.layers.advanced_activations.ELU()(conv9)
conv10 = Conv2D(num_mask_channels, 1, activation='sigmoid')(conv9)

model = Model(inputs=inputs, outputs=conv10)
```

```
    return model

def preprofunc(im):
    im = (im > 0.5)
    return im

model = get_unet()
# Compile model
model.compile(optimizer=Nadam(lr=1e-3, schedule_decay=lr_decay),
              loss=jaccard_coef_loss,
              metrics=['binary_crossentropy', jaccard_coef_int])


# Image Data Generator
# We create two instances with the same arguments
image_datagen = ImageDataGenerator(featurewise_center=True,
                     featurewise_std_normalization=True,
                     rotation_range=90,
                     width_shift_range=0.2,
                     height_shift_range=0.2,
                     zoom_range=0.2,
                     horizontal_flip = True)
mask_datagen = ImageDataGenerator(rotation_range=90,
                     width_shift_range=0.2,
                     height_shift_range=0.2,
                     zoom_range=0.2,
                     horizontal_flip = True,
                     preprocessing_function = preprofunc)

# Provide the same seed and keyword arguments to the fit and flow methods
seed = 1

if channels == 1:
    X_train = X_train.reshape(-1,128,128,1)
    X_test = X_test.reshape(-1,128,128,1)
y_train = y_train.reshape(-1,128,128,1)
y_test = y_test.reshape(-1,128,128,1)

nn = X_train.shape[0]
train_nn = int(nn * 0.9)
X_train = X_train[0:train_nn,:,:,:]
X_val = X_train[train_nn:,:,:,:]
y_train = y_train[0:train_nn,:,:,:]
y_val = y_train[train_nn:,:,:,:]

image_datagen.fit(X_train, augment=True, seed=seed)
mask_datagen.fit(y_train, augment=True, seed=seed)
```

```
image_generator = image_datagen.flow(
                    X_train,
                    seed=seed,
                    batch_size=BATCH_SIZE)
mask_generator = mask_datagen.flow(
                    y_train,
                    seed=seed,
                    batch_size=BATCH_SIZE)

image_generator_val = image_datagen.flow(
                        X_val,
                        seed=seed,
                        batch_size=BATCH_SIZE)
mask_generator_val = mask_datagen.flow(
                        y_val,
                        seed=seed,
                        batch_size=BATCH_SIZE)

# Combine generators into one which yields image and masks
train_generator = zip(image_generator, mask_generator)
validation_generator = zip(image_generator_val, mask_generator_val)

print("Finished data generator...")


# # # # # # # # # # # # # # # # # #

history = model.fit_generator(train_generator,
                              steps_per_epoch=NUM_STEPS,
                              epochs=NUM_EPOCHS,
                              validation_data=validation_generator,
                              validation_steps=10)

# # # Visualize Losses

bc = history.history['binary_crossentropy']
val_bc = history.history['val_binary_crossentropy']
jac = history.history['jaccard_coef_int']
val_jac = history.history['val_jaccard_coef_int']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(bc) + 1)

plt.rcParams['figure.figsize'] = (7, 3);

plt.plot(epochs, bc, 'bo', label='Training BC')
plt.plot(epochs, val_bc, 'b', label='Validation BC')
```

```
plt.title('Training and validation binary crossentropy')
plt.legend()

plt.figure()

plt.plot(epochs, jac, 'bo', label='Training Jaccard')
plt.plot(epochs, val_jac, 'b', label='Validation Jaccard')
plt.title('Training and validation Jaccard score')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()

X_example_train = image_generator[0]
y_example_train = mask_generator[0]
y_example_pred_train = model.predict(X_example_train)
X_example_test = image_generator_val[0]
y_example_test = mask_generator_val[0]
y_example_pred_test = model.predict(X_example_test)

def check_results(i, train=True):

    if train:
        X_example = X_example_train
        y_example = y_example_train
        y_example_pred = y_example_pred_train
    else:
        X_example = X_example_test
        y_example = y_example_test
        y_example_pred = y_example_pred_test

    fig, axs = plt.subplots(2,2, figsize=(8,8))
    axs = axs.flat
    if channels < 20:
        axs[0].imshow(X_example[i,:,:,0])
    else:
        axs[0].imshow(X_example[i,:,:,18])
    axs[2].imshow(y_example[i,:,:,0]); axs[2].set_title("True Mask")
    axs[3].imshow((y_example_pred[i,:,:,0] > .5)); axs[3].set_title("Pred. Mask")
    plt.show()

check_results(1, train=False)
```

```
check_results(2, train=False)
check_results(3, train=False)
check_results(4, train=False)
check_results(5, train=False)
check_results(6, train=False)

def evaluate_model():
    image_test_datagen = ImageDataGenerator(featurewise_center=True,
                         featurewise_std_normalization=True)

    image_test_datagen.fit(X_test, augment=True, seed=seed)

    image_test_generator = image_test_datagen.flow(
                        X_test, y_test,
                        seed=seed,
                        batch_size=X_test.shape[0])

    loss, bin_cross_entropy, jaccard = model.evaluate(image_test_generator[0][0],
                                                      image_test_generator[0][1],
                                                      verbose=0)
    print("Test Loss:", loss)
    print("Test Binary Cross Entropy:", bin_cross_entropy)
    print("Test Jaccard Score:", jaccard)
    print()

    image_train_datagen = ImageDataGenerator(featurewise_center=True,
                         featurewise_std_normalization=True)

    image_train_datagen.fit(X_train, augment=True, seed=seed)

    image_train_generator = image_train_datagen.flow(
                        X_train, y_train,
                        seed=seed,
                        batch_size=X_train.shape[0])

    loss,bin_cross_entropy,jaccard = model.evaluate(image_train_generator[0][0],
                                                    image_train_generator[0][1],
                                                    verbose=0)
    print("Train Loss:", loss)
    print("Train Binary Cross Entropy:", bin_cross_entropy)
    print("Train Jaccard Score:", jaccard)

model_str = 'model_' + str(feature) + '_' + str(channels)
model_str += 'channels_' + str(FEATURE_PRESENCE_MIN) + '_minfeature'
model_str += str(NUM_STEPS) + '_steps' + str(lr_decay) + '_lrDecay'
model_str += str(loss) + '_loss.hdf5'
model.save(model_str)
```

```
        try:
            evaluate_model()
        except:
            print("\nEvaluate Model Failed\n")

        return 0
```

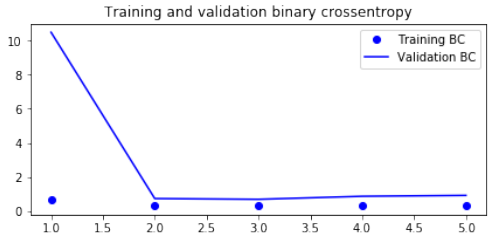
4 Results

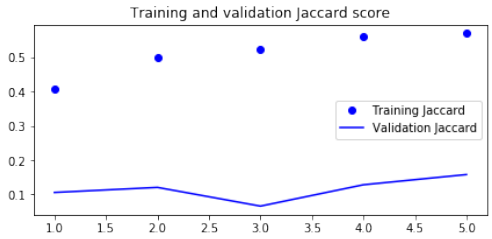
4.1 Preliminary Results

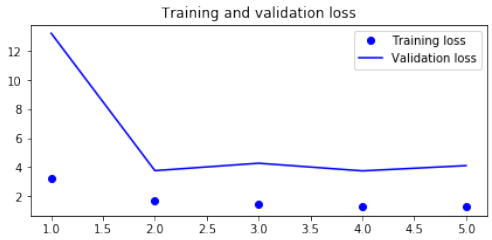
Here are some results from the function call. We ran these for 5 epochs, 200 steps per epoch, and batch size of 128. These are with learning rate decay, 20 channels, and on three features: buildings, roads, and trees. We tested on the full dataset (`FEATURE_PRESENCE_MIN` = 0) or only a subset where the features are present (`FEATURE_PRESENCE_MIN` = 0.04).

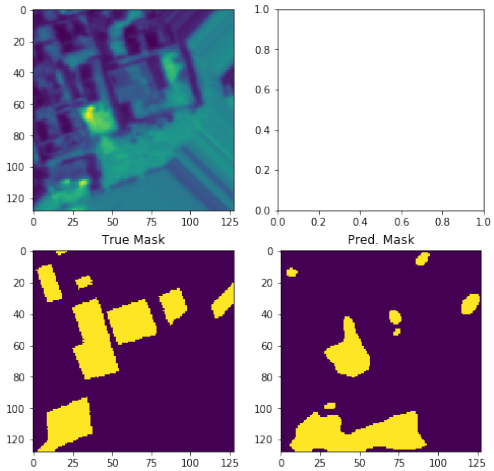
In these results, the visualizations are on the test set.

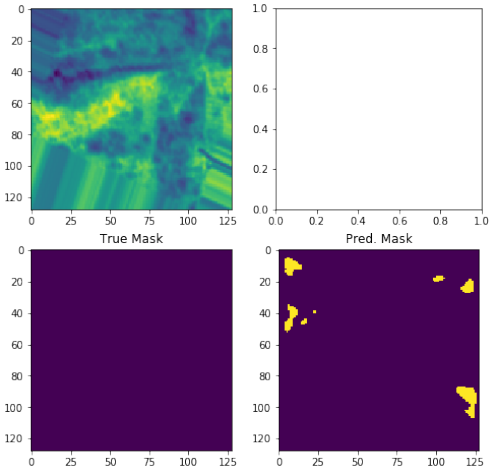
```
In [5]: do_model(feature='buildings', channels=20, FEATURE_PRESENCE_MIN = 0)

Starting...
Done data processing...


/usr/share/anaconda3/lib/python3.6/site-packages/keras_preprocessing/image.py:1358: UserWarning:
  ' channels).')
/usr/share/anaconda3/lib/python3.6/site-packages/keras_preprocessing/image.py:1643: UserWarning:
  str(self.x.shape[channels_axis]) + ' channels).')
/usr/share/anaconda3/lib/python3.6/site-packages/keras_preprocessing/image.py:1643: UserWarning:
  str(self.x.shape[channels_axis]) + ' channels).')


Finished data generator nonsense...
Epoch 1/5
200/200 [==============================] - 636s 3s/step - loss: 3.2143 - binary_crossentropy: 0.
Epoch 2/5
200/200 [==============================] - 617s 3s/step - loss: 1.6627 - binary_crossentropy: 0.
Epoch 3/5
200/200 [==============================] - 620s 3s/step - loss: 1.4558 - binary_crossentropy: 0.
Epoch 4/5
200/200 [==============================] - 618s 3s/step - loss: 1.3030 - binary_crossentropy: 0.
Epoch 5/5
200/200 [==============================] - 623s 3s/step - loss: 1.2537 - binary_crossentropy: 0.
```

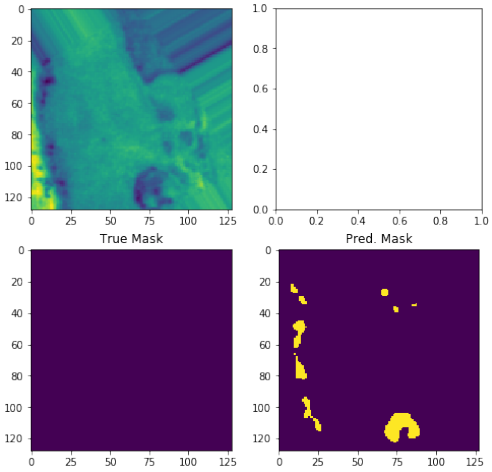

Training and validation binary crossentropy
Training BC
Validation BC

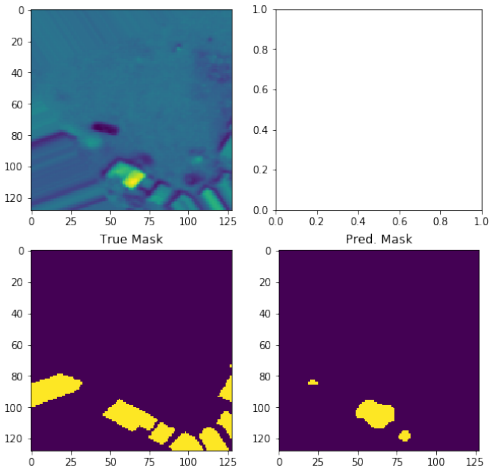

Training and validation Jaccard score
Training Jaccard
Validation Jaccard

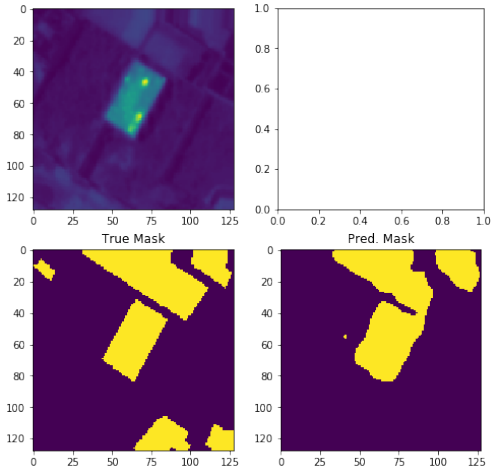

Training and validation loss
Training loss
Validation loss

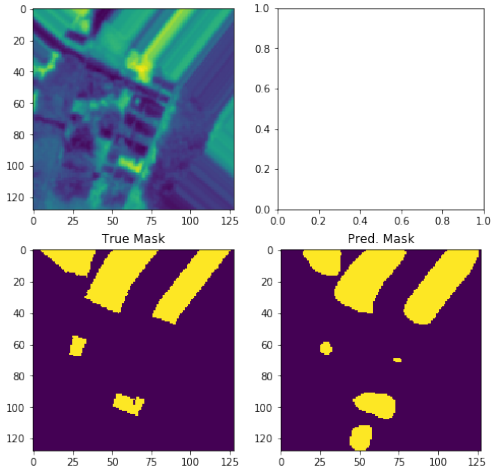

True Mask
Pred. Mask


True Mask
Pred. Mask


True Mask
Pred. Mask

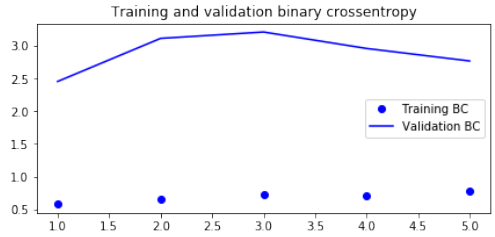

True Mask
Pred. Mask

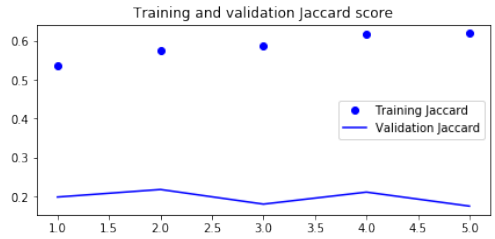


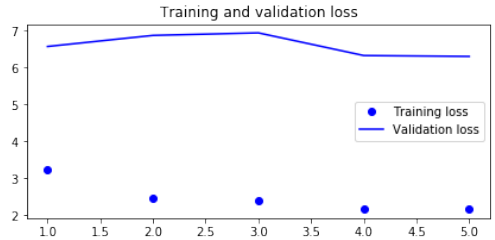


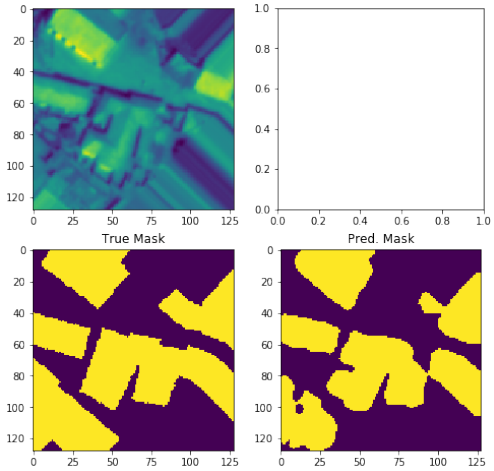

```
/usr/share/anaconda3/lib/python3.6/site-packages/keras_preprocessing/image.py:1358: UserWarning:
  ' channels).')


Test Loss: 4.191045435526037
Test Binary Cross Entropy: 0.7361179798521021
Test Jaccard Score: 0.06249437893750847

Train Loss: 2.4258285715985637
Train Binary Cross Entropy: 0.34166787132851295
Train Jaccard Score: 0.38522087782120334
```

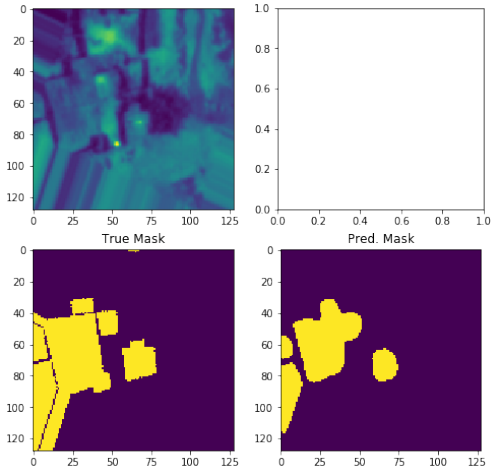
```
Out[5]: 0

In [13]: do_model(feature='buildings', channels=20, FEATURE_PRESENCE_MIN = 0.04)

Starting...
Done data processing...
Finished data generator nonsense...
Epoch 1/5
200/200 [==============================] - 603s 3s/step - loss: 3.2305 - binary_crossentropy: 0.
Epoch 2/5
200/200 [==============================] - 597s 3s/step - loss: 2.4758 - binary_crossentropy: 0.
Epoch 3/5
200/200 [==============================] - 596s 3s/step - loss: 2.3844 - binary_crossentropy: 0.
Epoch 4/5
200/200 [==============================] - 590s 3s/step - loss: 2.1475 - binary_crossentropy: 0.
Epoch 5/5
200/200 [==============================] - 597s 3s/step - loss: 2.1460 - binary_crossentropy: 0.
```

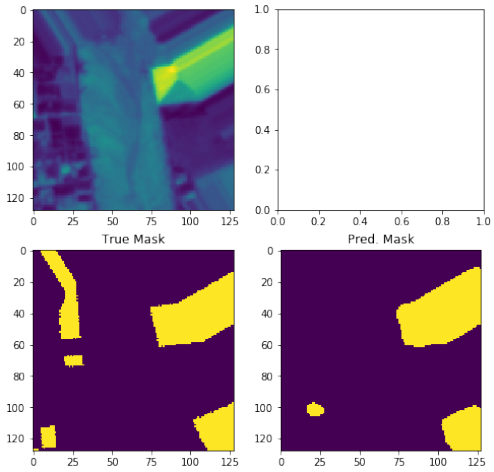


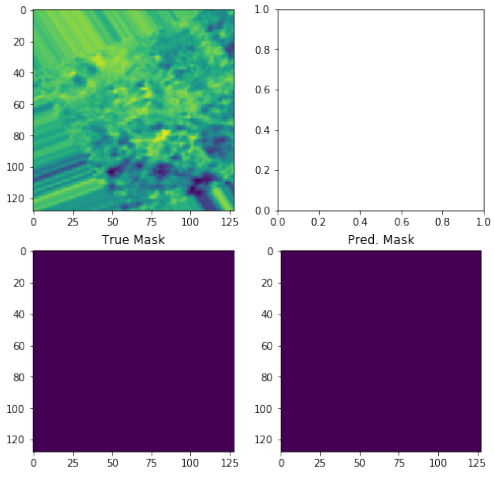


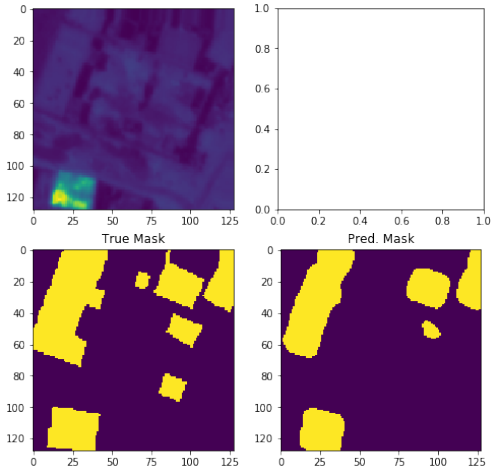


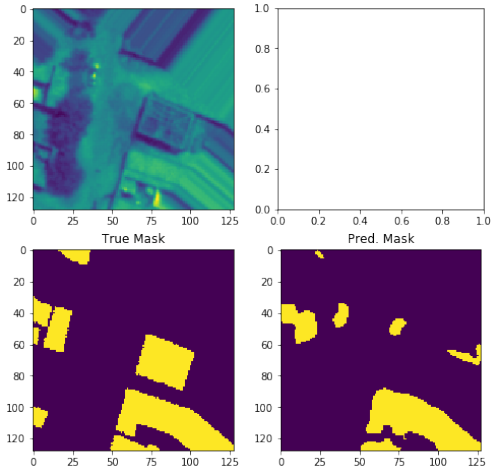

True Mask
Pred. Mask


True Mask
Pred. Mask


True Mask
Pred. Mask


True Mask
Pred. Mask


True Mask
Pred. Mask




```
/usr/share/anaconda3/lib/python3.6/site-packages/keras_preprocessing/image.py:1358: UserWarning:
  ' channels).')


Test Loss: 5.9444373081185224
Test Binary Cross Entropy: 2.223455414147276
Test Jaccard Score: 0.07687959804240918

Train Loss: 1.9869081195402536
Train Binary Cross Entropy: 0.6672162292691399
Train Jaccard Score: 0.6345208949129534
```

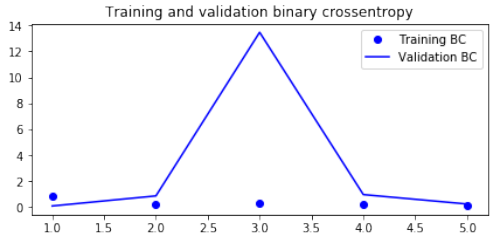
```
Out[13]: 0
```

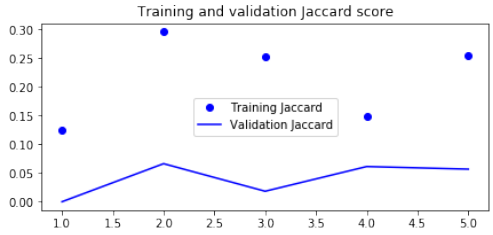
Looking at the buildings result on the whole dataset compared to the FEATURE_PRESENCE_MIN = 0.04, the quantitative results are clearly better for the subsetted data, as we would expect, with a significantly higher train Jaccard score and lower overall loss for the train and test sets compared to the whole dataset.

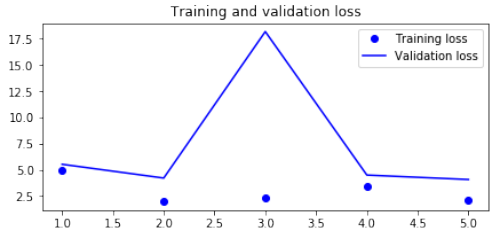
Qualitatively assessing the results, the predicted masks for the subset look better than the predicted masks for the entire dataset for the samples that were plotted. The second model looks like it captures features in the true mask better than the first model.

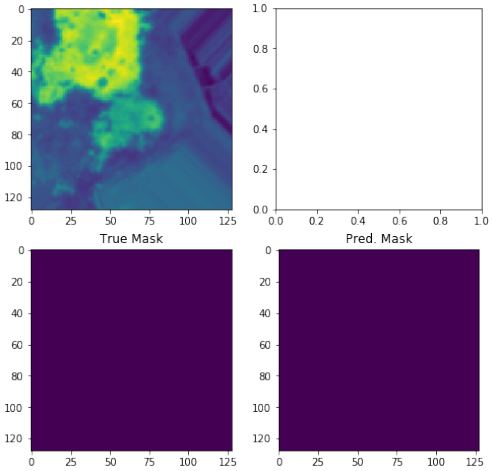
```
In [6]: do_model(feature='road', channels=20, FEATURE_PRESENCE_MIN = 0)

Starting...
Done data processing...
Finished data generator nonsense...
Epoch 1/5
200/200 [==============================] - 626s 3s/step - loss: 4.9573 - binary_crossentropy: 0.
Epoch 2/5
200/200 [==============================] - 626s 3s/step - loss: 1.9557 - binary_crossentropy: 0.
```

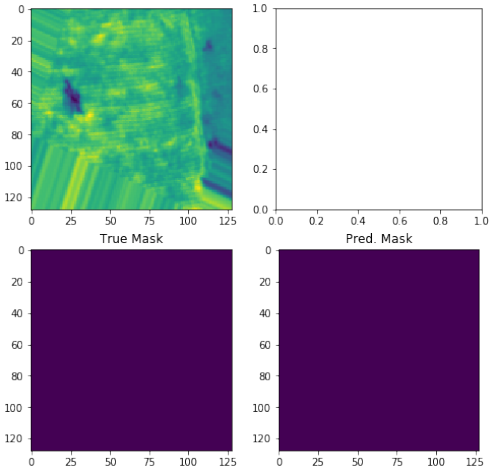
```
Epoch 3/5
200/200 [==============================] - 619s 3s/step - loss: 2.3802 - binary_crossentropy: 0.
Epoch 4/5
200/200 [==============================] - 618s 3s/step - loss: 3.4108 - binary_crossentropy: 0.
Epoch 5/5
200/200 [==============================] - 622s 3s/step - loss: 2.1667 - binary_crossentropy: 0.
```

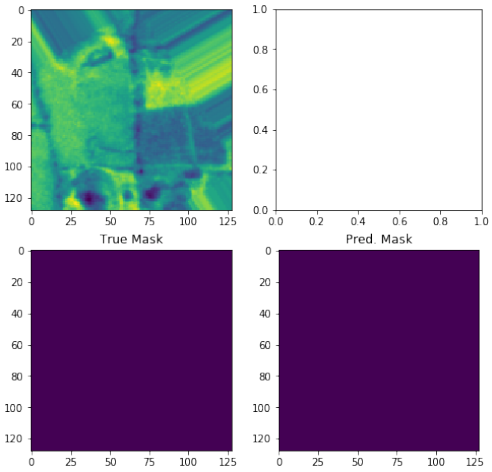


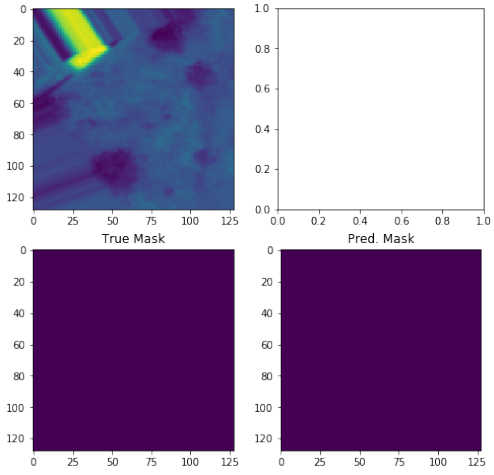


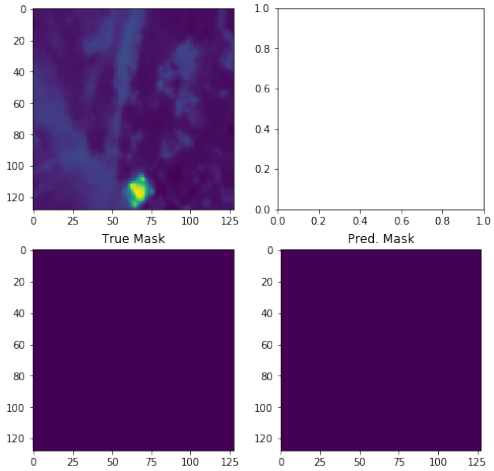


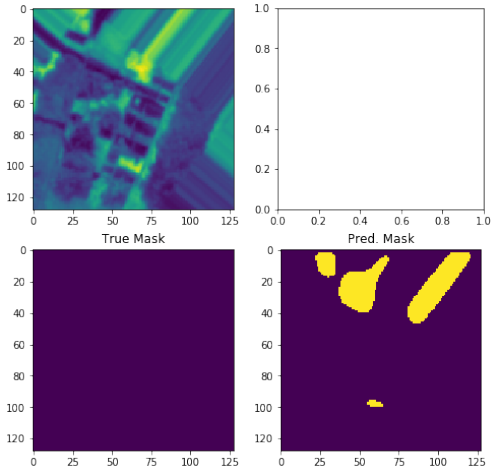

True Mask
Pred. Mask


True Mask
Pred. Mask


True Mask
Pred. Mask


True Mask
Pred. Mask


True Mask
Pred. Mask



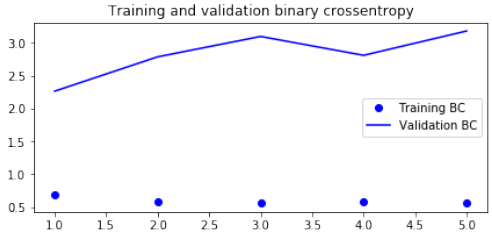

```
Test Loss: 8.14406257547358
Test Binary Cross Entropy: 0.12729044079590307
Test Jaccard Score: 0.12676411290323839

Train Loss: 6.084402412476761
Train Binary Cross Entropy: 0.18308775986732054
Train Jaccard Score: 0.10747224844603459
```

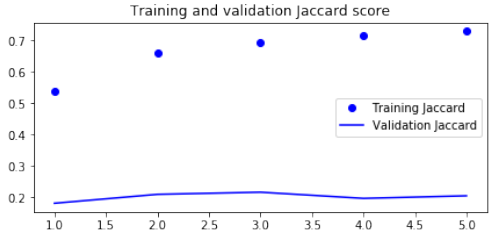
```
Out[6]: 0
```

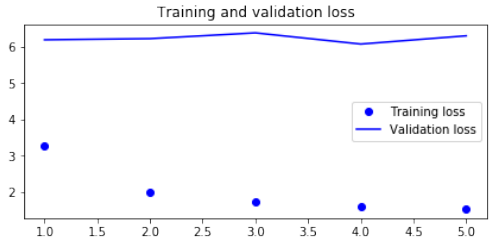
```
In [14]: do_model(feature='road', channels=20, FEATURE_PRESENCE_MIN = 0.04)
```

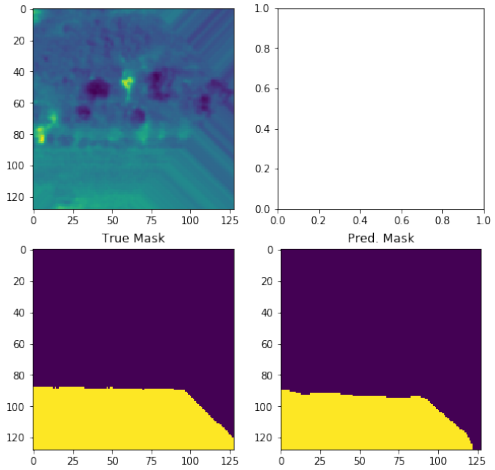
```
Starting...
Done data processing...


/usr/share/anaconda3/lib/python3.6/site-packages/keras_preprocessing/image.py:1358: UserWarning:
  ' channels).')
/usr/share/anaconda3/lib/python3.6/site-packages/keras_preprocessing/image.py:1643: UserWarning:
  str(self.x.shape[channels_axis]) + ' channels).')
/usr/share/anaconda3/lib/python3.6/site-packages/keras_preprocessing/image.py:1643: UserWarning:
  str(self.x.shape[channels_axis]) + ' channels).')


Finished data generator nonsense...
Epoch 1/5
200/200 [==============================] - 596s 3s/step - loss: 3.2690 - binary_crossentropy: 0.
Epoch 2/5
200/200 [==============================] - 579s 3s/step - loss: 1.9710 - binary_crossentropy: 0.
Epoch 3/5
200/200 [==============================] - 586s 3s/step - loss: 1.7244 - binary_crossentropy: 0.
```

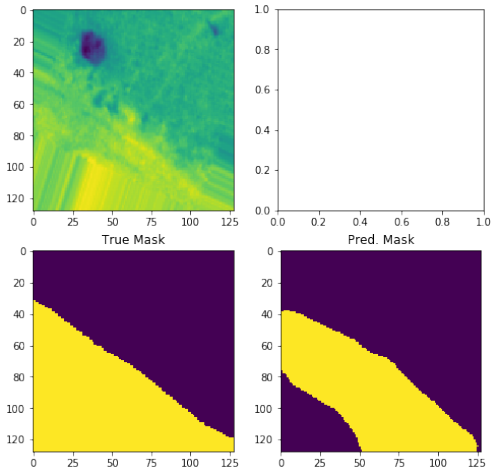
```
Epoch 4/5
200/200 [==============================] - 588s 3s/step - loss: 1.6009 - binary_crossentropy: 0.
Epoch 5/5
200/200 [==============================] - 588s 3s/step - loss: 1.5114 - binary_crossentropy: 0.
```

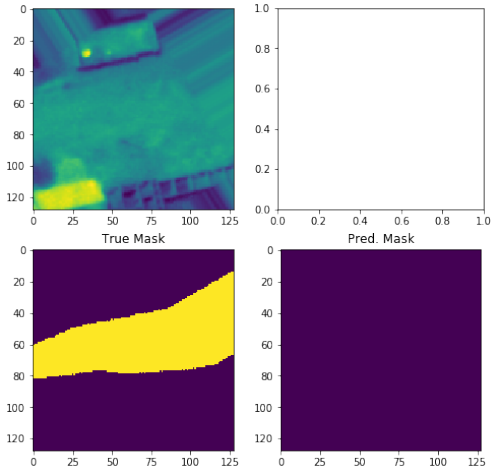


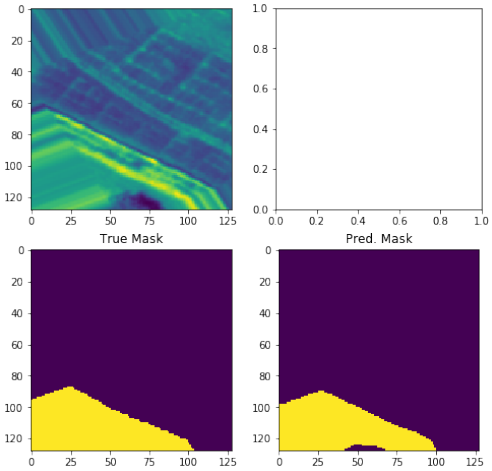


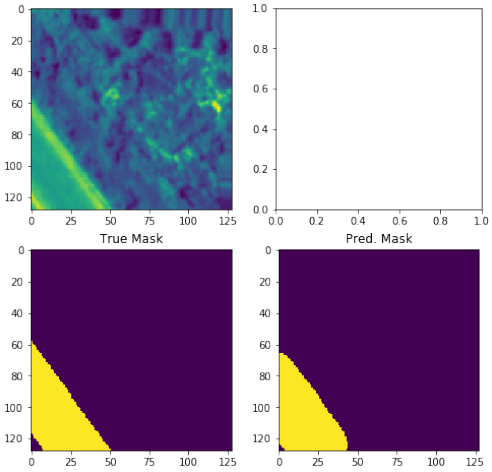


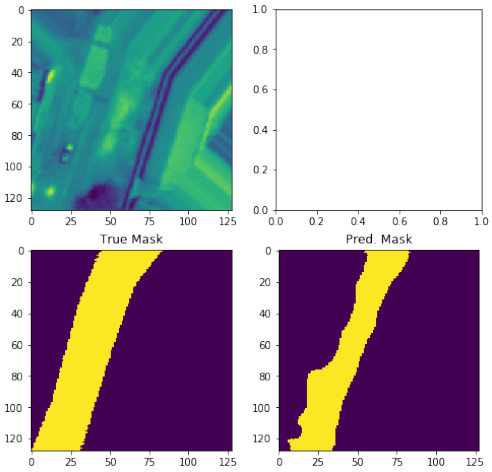



True Mask
Pred. Mask


True Mask
Pred. Mask


True Mask
Pred. Mask

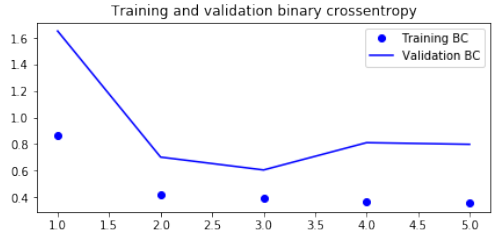

True Mask
Pred. Mask

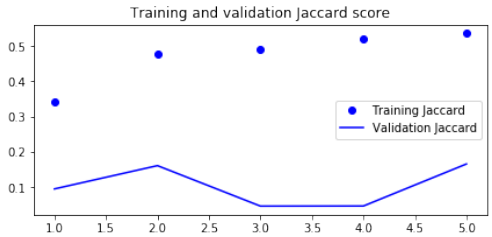

True Mask
Pred. Mask

```
/usr/share/anaconda3/lib/python3.6/site-packages/keras_preprocessing/image.py:1358: UserWarning:
  ' channels).')


Test Loss: 1.5624375263849895
Test Binary Cross Entropy: 0.6484147985776265
Test Jaccard Score: 0.71253875096639

Train Loss: 1.016711446213573
Train Binary Cross Entropy: 0.3562162139594928
Train Jaccard Score: 0.8007569367800973


Out[14]: 0
```

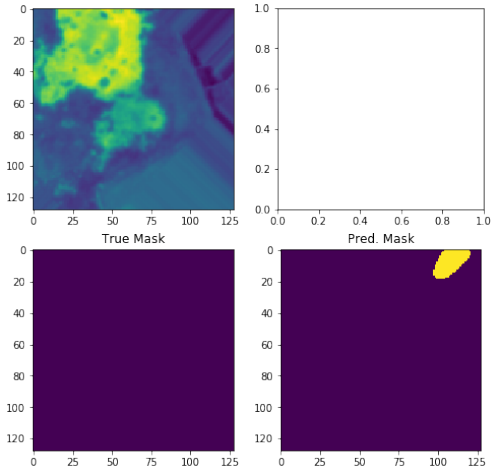
For roads, the models overfit less and perform much better on the train and test cases. In this case, the test Jaccard score is very low for the full dataset and very high for the subset. This is probably because roads are a very uncommon class, so when we only focus on tiles with roads in them, the model performs much better. This suggests that if we could train a CNN classifier to classify whether a tile has roads or not in it to reduce the number of input images, then feed it to a model that is trained only on tiles that definitely have roads in them. Thus, strategies that we've learned for classification with infrequent classes would be ideal for this additional model.

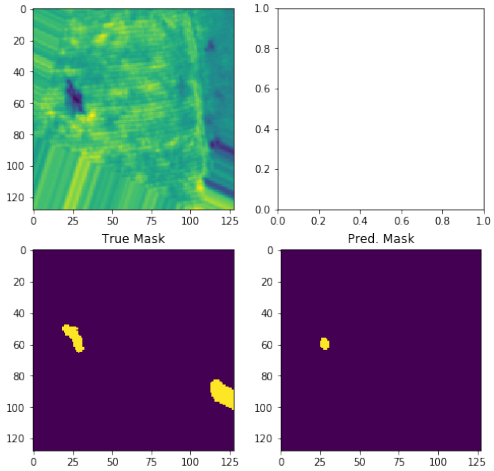
```
In [7]: do_model(feature='trees', channels=20, FEATURE_PRESENCE_MIN = 0)

Starting...
Done data processing...
Finished data generator nonsense...
Epoch 1/5
200/200 [==============================] - 625s 3s/step - loss: 3.6619 - binary_crossentropy: 0.
Epoch 2/5
200/200 [==============================] - 616s 3s/step - loss: 1.7137 - binary_crossentropy: 0.
Epoch 3/5
200/200 [==============================] - 622s 3s/step - loss: 1.5565 - binary_crossentropy: 0.
Epoch 4/5
200/200 [==============================] - 623s 3s/step - loss: 1.3835 - binary_crossentropy: 0.
Epoch 5/5
200/200 [==============================] - 616s 3s/step - loss: 1.3030 - binary_crossentropy: 0.
```

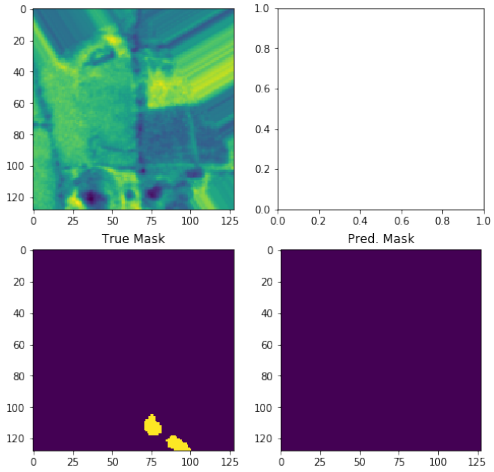


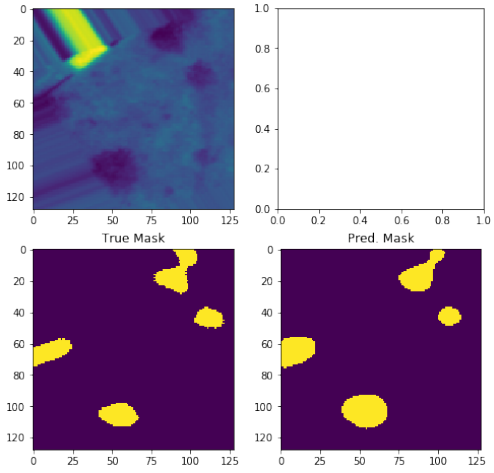

Training and validation Jaccard score
Training Jaccard
Validation Jaccard

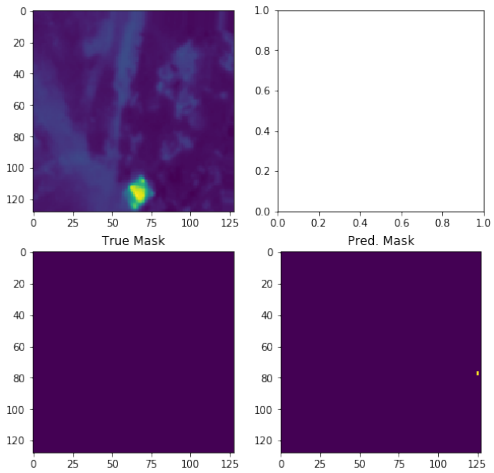

Training and validation loss
Training loss
Validation loss

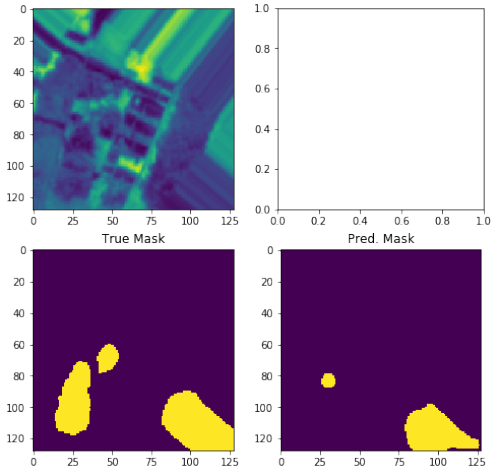

True Mask
Pred. Mask


True Mask
Pred. Mask


True Mask
Pred. Mask


True Mask
Pred. Mask



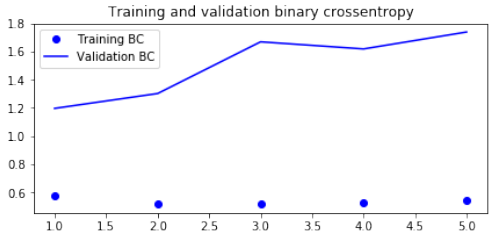


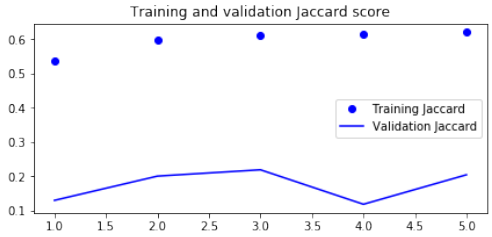

```
Test Loss: 5.741649586667297
Test Binary Cross Entropy: 3.5836623714816187
Test Jaccard Score: 0.1926655525802284

Train Loss: 1.1535529845019035
Train Binary Cross Entropy: 0.251483828951708
Train Jaccard Score: 0.5495954941564867
```

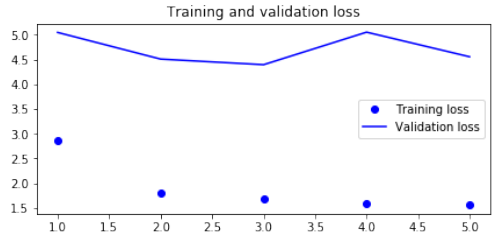
```
Out[7]: 0
```

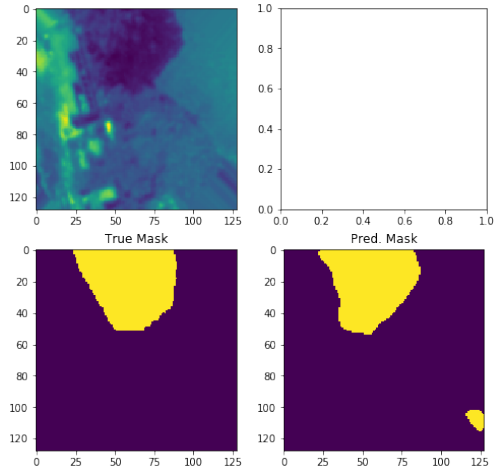
```
In [15]: do_model(feature='trees', channels=20, FEATURE_PRESENCE_MIN = 0.04)
```

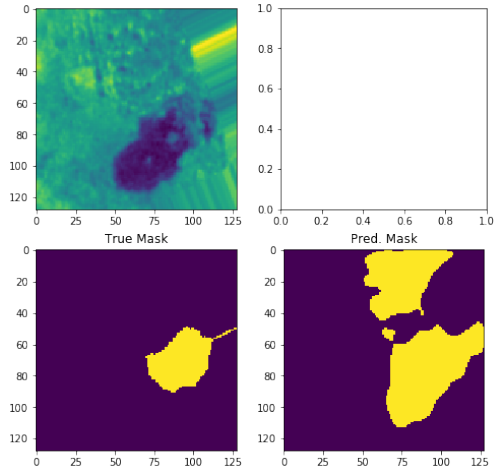
```
Starting...
Done data processing...


/usr/share/anaconda3/lib/python3.6/site-packages/keras_preprocessing/image.py:1358: UserWarning:
  ' channels).')
/usr/share/anaconda3/lib/python3.6/site-packages/keras_preprocessing/image.py:1643: UserWarning:
  str(self.x.shape[channels_axis]) + ' channels).')
/usr/share/anaconda3/lib/python3.6/site-packages/keras_preprocessing/image.py:1643: UserWarning:
  str(self.x.shape[channels_axis]) + ' channels).')


Finished data generator nonsense...
Epoch 1/5
200/200 [==============================] - 621s 3s/step - loss: 2.8609 - binary_crossentropy: 0.
Epoch 2/5
200/200 [==============================] - 613s 3s/step - loss: 1.8198 - binary_crossentropy: 0.
Epoch 3/5
200/200 [==============================] - 628s 3s/step - loss: 1.6777 - binary_crossentropy: 0.
Epoch 4/5
200/200 [==============================] - 611s 3s/step - loss: 1.6052 - binary_crossentropy: 0.
Epoch 5/5
200/200 [==============================] - 619s 3s/step - loss: 1.5606 - binary_crossentropy: 0.
```

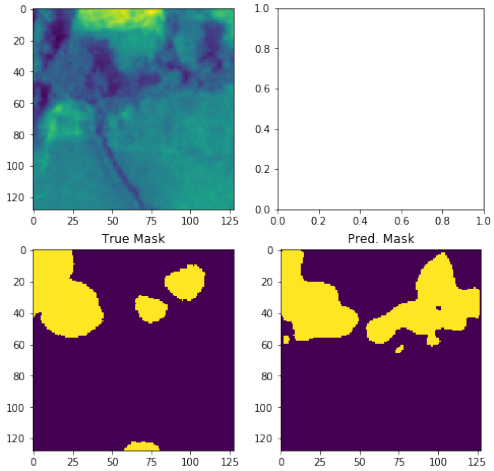


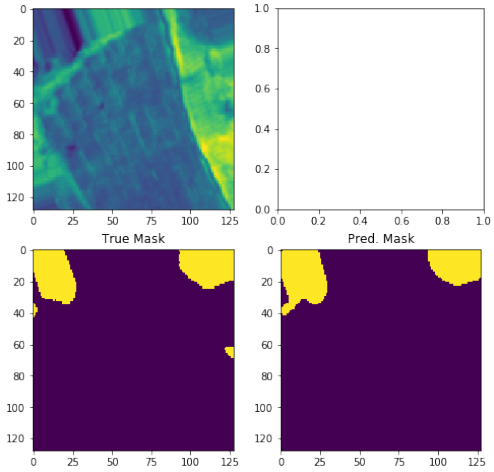


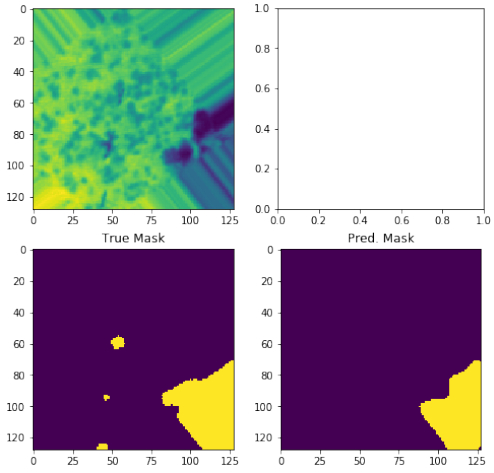

Training and validation loss
Training loss
Validation loss

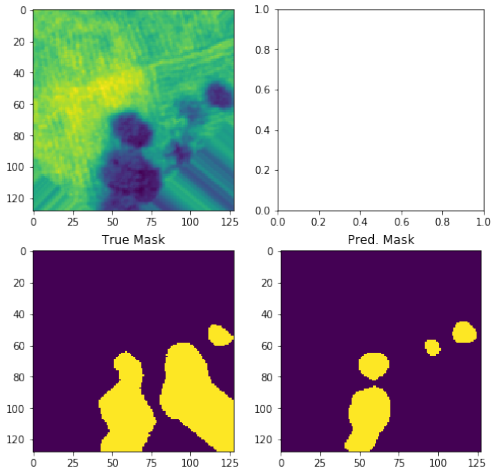

True Mask
Pred. Mask


True Mask
Pred. Mask


True Mask
Pred. Mask


True Mask
Pred. Mask

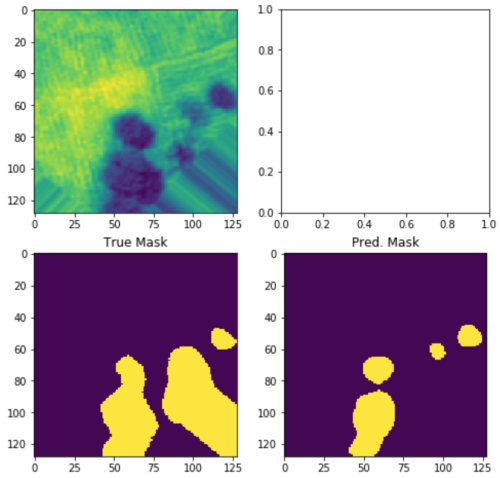

True Mask
Pred. Mask




```
/usr/share/anaconda3/lib/python3.6/site-packages/keras_preprocessing/image.py:1358: UserWarning:
  ' channels).')


Test Loss: 8.717169186495996
Test Binary Cross Entropy: 0.9780846594840057
Test Jaccard Score: 0.003506079853274943

Train Loss: 1.5632958563723305
Train Binary Cross Entropy: 0.5271791709485905
Train Jaccard Score: 0.6160536169081695
```

Out[15]: 0

For trees, our models overfit slightly more than with the roads. Looking at the visualizations, the predicted masks miss some key areas of the true masks. Looking at the last visualization above, it looks like the predicted mask missed something that could not be a tree:



By visual inspection, the missing bit does not look like a tree, so it is understandable that the model might have missed it. We could not determine that that was a tree, so it's difficult for the model to predict it.

4.2 Summary of Results

We trained a variety of models to perform semantic segmentation on hyperspectral satellite imagery. We focused on three surface classes: buildings, trees, and roads. We evaluated our models in terms of the Jaccard score, which is defined as the intersection over union of the predicted segmentation masks and ground-truth segmentation masks. We analyzed the sensitivity of our results to different model hyperparameters and training strategies. In some cases, we achieved performance comparable to top submissions to the 2017 DSTL Kaggle competition.

4.2.1 Note on Initializing Model Weights

For some surface classes, we found that model performance varied considerably with the random initialization of the model weights. This indicates that the associated loss functions are complex and hence that the optimization is difficult to tune. The same behavior was also observed during the Kaggle competition in 2017. It was problematic enough at the time that, just two days before the end of the competition, one team (which ended up taking third place) published a set of model weights for others to use for initializing their models. This sparked a debate about the ethics of sharing insights near the end of competitions, and the published model weights have since been removed. Naturally, we trained all of our models with random weight initializations and therefore do not expect to replicate the results of the winning submissions. In some cases, however, we do obtain comparable results.

4.2.2 Tables

TABLE 1: Dependence of Jaccard score on the number of image channels used, by surface class and minimum feature presence.

	1 channel (train | test)		3 channels (train | test)		20 channels (train | test)	
FEAT_MIN:	**0**	**0.04**	**0**	**0.04**	**0**	**0.04**
Buildings	0.41 | 0.43	0.542 | 0.537	0.587 | 0.230	0.598 | 0.058	0.385 | 0.062	0.635 | 0.077
Roads	0.039 | 0.035	0.727 | 0.727	0.139 | 0.018	0.765 | 0.474	0.128 | 0.107	0.800 | 0.713
Trees	0.224 | 0.207	0.479 | 0.446	0.259 | 0.0004	0.467 | 0.307	0.550 | 0.193	0.616 | 0.004

200 Steps, 5 Epochs, Batch size 128

TABLE 2: Dependence of Jaccard score on loss function used (Jaccard loss, binary cross-entropy, or both) for 20-channel images and minimum feature presence of 0.04, by surface class.

	Jaccard loss only (train | test)	Binary Cross-Entropy (train | test)	Both losses (train | test)
FEAT_MIN:	**0.04**	**0.04**	**0.04**
Buildings	0.683 | 0.235	0.651 | 0.226	0.635 | 0.077
Roads	0.774 | 0.688	0.716 | 0.663	0.800 | 0.713
Trees	0.654 | 0.007	0.643 | 0.003	0.616 | 0.004

200 Steps, 5 Epochs, Batch size 128; 20 channels

TABLE 3: Same as Table 2, except for 1-channel images.

	Jaccard loss only (train | test)	Binary Cross-Entropy (train | test)	Both losses (train | test)
FEAT_MIN:	**0.04**	**0.04**	**0.04**
Buildings	0.545 | 0.540	0.486 | 0.476	0.542 | 0.537
Roads	0.762 | 0.668	0.747 | 0.671	0.727 | 0.727
Trees	0.518 | 0.494	0.352 | 0.375	0.479 | 0.446

200 Steps, 5 Epochs, Batch size 128; 1 channel

TABLE 4: Dependence of Jaccard score on learning rate decay (on/off), by surface class and minimum feature presence.

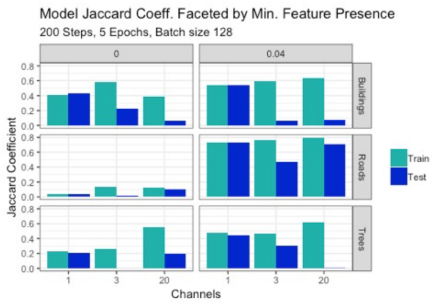
	No lr decay (train | test)		With 0.004 lr decay (train | test)	
FEAT_MIN:	**0**	**0.04**	**0**	**0.04**
Buildings	0.219 | 0.024	0.666 | 0.180	0.385 | 0.062	0.635 | 0.077
Roads	0.217 | 0.039	0.779 | 0.671	0.128 | 0.107	0.800 | 0.713
Trees	0.315 | 0.186	0.655 | 0.334	0.550 | 0.193	0.616 | 0.004

200 Steps, 5 Epochs, Batch size 128; 20 channels

4.3 Sensitivity Tests

We conducted several sensitivity tests to evaluate (1) the predictive power of different hyperspectral data products (i.e., panchromatic vs. RGB. vs. 16-band); (2) the effectiveness of different loss functions; (3) the impact of requiring surface classes to be present in the training data; (4) the influence of learning rate decay on model performance; and (5) different regularization strategies. For each sensitivity test, we trained a small ensemble of models on various surface classes (mainly trees, roads, and buildings) and compared Jaccard scores on train and test sets.

4.3.1 Minimum Feature Presence



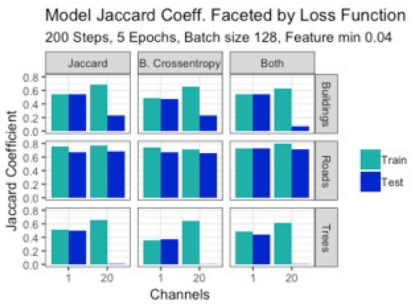
A key challenge in segmenting satellite imagery is that the surface classes are rarely balanced. For example, satellite imagery of an urban scene will generally contain far fewer road pixels than non-road pixels. One of the most important sensitivity tests we performed was therefore to alternate between showing our model all available training samples and showing it only training

samples that contained the surface class of interest. We parameterized this condition with a "minimum feature presence" parameter defined as the fraction of pixels in a training image that must belong to the surface class. For example, a minimum feature presence of 0.1 for buildings would mean training the model only on images for which at least 10% of pixels are buildings. We experimented with two values for this parameter: 0 and 0.04. As shown in Tables 1 and 4, we found that our models performed much better when trained and evaluated on the more balanced data, with a minimum feature presence of 4%. The effect was particularly strong for roads, where we achieved a peak Jaccard score on the test images of 0.83. In contrast, with a minimum feature presence of 0% (i.e., training and evaluating on all available images), our best Jaccard score for roads is 0.73. This is still comparable to the score of 0.80 achieved by the team that placed third in the competition. Our models for buildings and trees were also improved after imposing the minimum feature presence of 4%, but to a lesser degree than for roads.

4.3.2 Different Hyperspectral Data Products

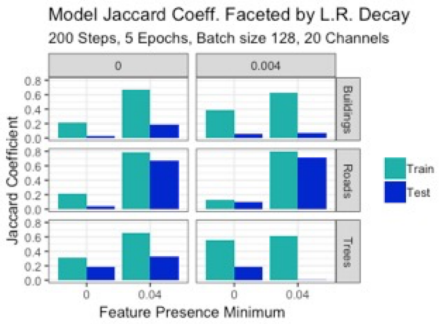
Another important sensitivity test involved using different combinations of hyperspectral data products to determine which are the most effective at predicting pixel classes. For this test, we trained an ensemble of models using (i) panchromatic images; (ii) RGB images; and (iii) the 20-channel panchromatic, RGB, multispectral, and SWIR datacubes created during preprocessing. This allowed us to test whether including additional spectral layers would help improve the classification models, or if it merely introduced noise from irrelevant spectral data. We found that in many cases, the full 20-channel images did in fact introduce noise; in those cases, the predictive power of the model decreased as the number of layers increased. The existence of this negative correlation between predictive power and number of layers was dependent on class, as was the strength of the relationship when it did exist. For example, our best model for segmenting buildings and trees used only the 1-channel panchromatic images. In contrast, adding the RGB and then 16-band data improved road prediction when no condition was placed on feature presence, but had no effect when the minimum feature presence was set to 4%.

4.3.3 Different Loss Functions



We tested three different loss functions: (i) Jaccard loss, (ii) binary crossentropy, and (iii) Jaccard loss plus binary crossentropy. The results are summarized in Tables 2 and 3. We found that while the binary crossentropy loss function tends to produce the worst results, which of the Jaccard loss or combined Jaccard and binary crossentropy loss functions produces the best results depends on surface class. For trees and buildings, using both 1-channel and 20-channel images, we found that the lone Jaccard loss performed best. In contrast, for roads we found that the combined Jaccard and binary crossentropy score performed best.

4.3.4 Learning Rate Decay



We also tested the model's performance by altering the value for learning rate decay in training. We tested two levels of learning rate decay: (i) 0 an (ii) 0.004. The purpose of learning rate decay is to allow for the model to move quickly from its initialization point to a range of "good" values for its parameters, but then for learning to slow so that the model can find the optimized value for each parameter without jumping over it. The result is summarized in table 4. We found that in general, the results of this test were inconclusive, and model performance was not significantly better for the models we trained with learning rate decay at 5 epochs. When feature presence minimum was set to 0, the model trained with a learning rate decay of 0.004 had a higher test Jaccard coefficient for all of the features. When the feature presence minimum was set to 4%, the values for the test Jaccard coefficient decreased significantly for both buildings and trees when we added a value for learning rate decay, but increased marginally for roads.

4.3.5 Different Regularization Strategies

Our standard U-Net model is regularized by Batch Normalization layers alone. Batch normalization serves the primary purpose of accelerating learning by reducing variance in the model's hidden layer values. However, in the process, it also adds noise to hidden layer activations, and this introduces some regularization to the optimization. As shown in Tables 1-4, we found that for many of our models, but especially those trained to segment trees, this regularization was insufficient to prevent overfitting. In these cases, training Jaccard scores were much higher than test Jaccard scores. The effect tends to be larger for models trained on 20-channel images, as much of the 20-channel data is effectively noise that prevents the model from generalizing to unfamiliar data. To test whether additional regularization would ameliorate the situation, we trained a new U-Net model with 50% dropout layers in each convolutional block of the expanding pathway to segment trees using the 20-channel data (with 0% minimum feature presence). We trained the model for 20 epochs with 400 steps per epoch and a batch size of 128, and found that the test Jaccard score was improved from 0.14 for the standard batch-normalized U-Net to 0.18 for the U-Net with batch normalization and dropout. We therefore believe that, with more time, we could improve our 1-channel tree segmentation U-Net by adding dropout layers to its expanding pathway.

5 Conclusion

This is an open area of research, thus improved solutions will continue to be created as more methods and novel architectures are tested and implemented. That being said, some key takeaways we found from our work include:

- Using a U-net architecture and image sub-setting (filtering for feature maps with features representing more than 5% of the image), accurate models for roads and standing water are possible. In both cases we achieved Jaccard scores of above 0.7 on test data. In both of these cases we used all 20 image channels.
- In some classes, it seems using all 20 channels was less effective than using just 1 or 3. This was especially true for the buildings class, which achieved a Jaccard score of 0.54 when using a single channel and image sub-setting.
- For both trees and buildings, as the number of channels increased, the test Jaccard scores decreased while the train Jaccard scores stayed relatively high and constant. This suggested the added noise from the additional channels may have been causing some overfitting. Trees especially did better with fewer channels.
- Using a decaying learning rate (Nadam) improved Jaccard scores for some classes, such as roads.
- In all cases, using image sub-setting by filtering for feature maps with features representing more than 5% of the image yielded significantly better scores, both Jaccard and BCE.
- Vehicles, both small and large, were especially difficult to train.

6 Sources

Kaggle Competition Page - https://www.kaggle.com/c/dstl-satellite-imagery-feature-detection/overview

Kaggle First Place Blog Post - http://blog.kaggle.com/2017/04/26/dstl-satellite-imagery-competition-1st-place-winners-interview-kyle-lee/

4th Place Blog Post - https://deepsense.ai/deep-learning-for-satellite-imagery-via-image-segmentation/

U-Nets Implementation Guide - https://www.depends-on-the-definition.com/unet-keras-segmenting-images/

Original U-Net Paper - https://arxiv.org/pdf/1505.04597.pdf

Data Tiling Sources - https://gist.github.com/KeremTurgutlu/68feb119c9dd148285be2e247267a203 (for nearest neighbor interpolation) - https://stackoverflow.com/questions/16856788/slice-2d-array-into-smaller-2d-arrays (function to tile 2d array into smaller 2d arrays)

Model Sources: - https://github.com/ternaus/kaggle_dstl_submission (3rd place winner code, where we got the model that we're currently using)

Mask Preprocessing sources - https://www.kaggle.com/iglovikov/jaccard-polygons-polygons-mask-polygons (converting WKT multipolygon data to boolean numpy array)