Fully Convolutional Network in Land-Water Classification

CSCI 8980 Final Project

Dec 13, 2017

Akhil Bhargava

Table of Contents

Introduction

Dataset

The dataset used contains images of the globe taken by NASA's moderate-resolution imaging spectroradiometer (MODIS)[1]. These images were taken daily, and subsequently extracted to gather images that contained lakes and rivers from the dates between 2/18/2000 - 3/18/2000. Using the temporal context of an eleven day timeframe, the images at the same location are combined to generate a new cleaned image. MODIS collects 500 m sq. resolution data using seven bands.

Number	**Bandwidth (nm)**	**Name**
Band 1	620 - 670	Red
Band 2	841 - 876	Near Infrared (NIR)
Band 3	459 - 479	Blue
Band 4	545 - 565	Green
Band 5	1230 - 1250	Near Infrared (NIR)
Band 6	1628 - 1652	Shortwave Infrared (SWIR)
Band 7	2105 - 2155	Shortwave Infrared (SWIR)

Table 1: Band Description

Using the images, two datasets are procured: One containing the original images of the lakes conserving its spatial information (ImageLevel_Dataset), and the other containing a pixel's class and its corresponding values for the 7 bands (Pixelwise_Dataset). The data amounts to 915 lakes containing 1852618 pixels.

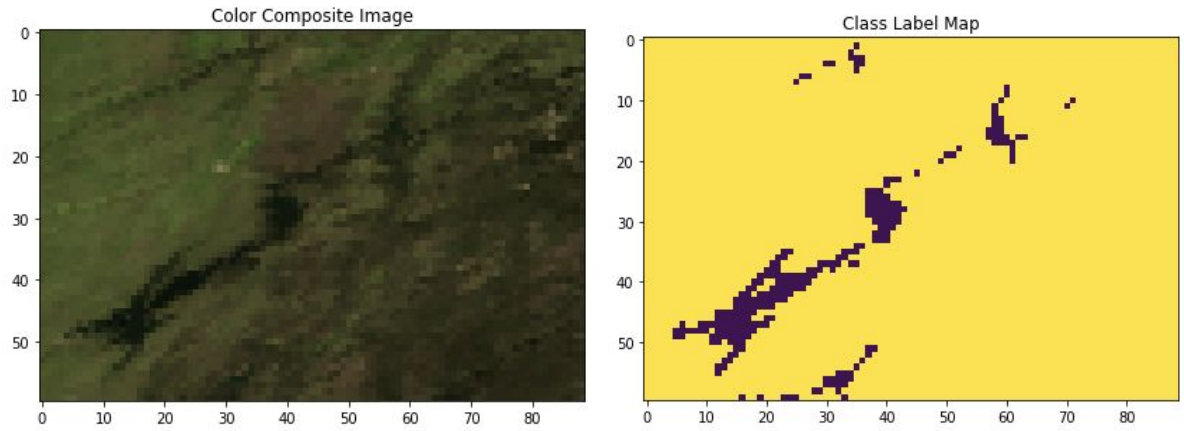

Figure 1 Lake and Labels: Using the RGB bands, the color composite image of the lake is shown along with the class label map. Purple indicates water, and yellow indicates land.

Goal

The goal of the project focuses on utilizing a Fully Connected Neural Network (FCN) in order to perform dense pixelwise classification using the ImageLevel_Dataset using only convolutional and deconvolutional layers. Generation of the model will allow the ability to answer three main questions:

1). Will an FCN perform better than a naive classifier (choosing all pixels are land)?
For any model to be valid, it must be informative. The dataset contains two classes, land and water. The FCN should perform better than the naive classifier which would assume all pixels are land.

2). Is there a difference using the RGB bands versus all 7 bands in a FCN?
Most Convolutional Neural Network research focuses on image segmentation utilizing the Red, blue, and green bands. It would be interesting to determine whether or not the inclusion of infrared and shortwave infrared bands will aid in the classification performance in the FCN.

3). Is spatial context a boon to pixelwise classification?
The purpose of the FCN is to capitalize on the spatial correlation that is inherent in land and water. In theory, if a pixel is surrounded by water, it is most likely of class water as well. In order to test this, a simple Logistic Regression model was implemented on the Pixelwise_Dataset. If spatial context truly is a boon, the classification performance of the FCN should fare better than the Logistic Regression model.

Fully Convolutional Network Methodology

Input

Varying Image Size Issue

Utilizing Keras with tensorflow as its backend within Python, the required input shape for the features is: the *number of images, width of image, height of image, number of bands*. For example, if the dataset contained one-thousand 512 * 512 images, the corresponding numpy 4D array would be (1000, 512, 512, 7). The label array would be: *number of images, width * height, number of classes,* where the class is an encoded binarization. Following the original example, this would equate to a 3D array (1000, 262144, 2). The issue with the input requirements is the varying dimensions of the images within the ImageLevel_Dataset. The images can range from 12 * 28 to 132 * 132. As a result, it was decided to select patches of width *w* and length *l* of images. One important note to consider is the training, validation, and test set split. It was determined that specific lakes should be first split into the training, validation and test sets, and then performing the patch-creation for each lake. The idea would ensure that there was no bias in training the FCN on a specific image of the lake or river and subsequently testing on the same image.

Patch Creation

A patch creation function was implemented in order to generate patches of a lake with a specified *w* and *l*. *Code can be found in the appendix.*

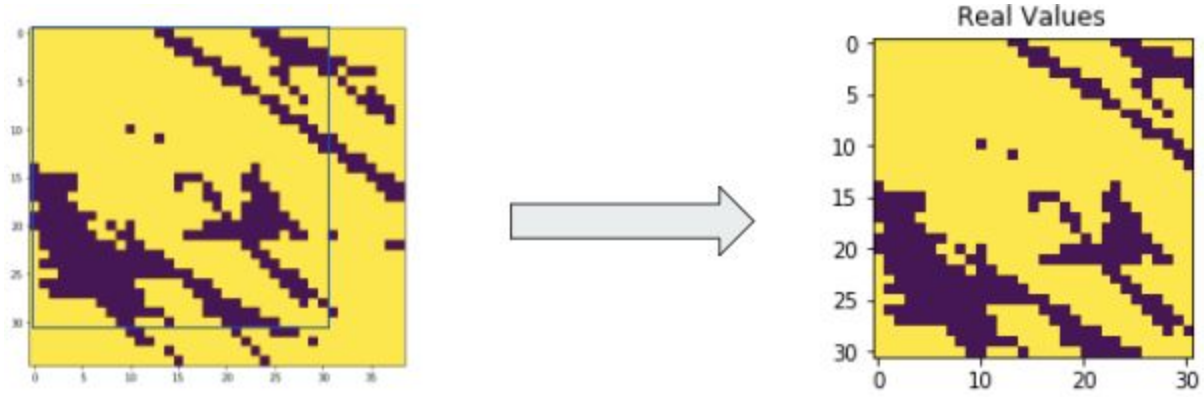

Figure 2 Patch Depiction: An image of size 38 * 36 was used as an input to output a single patch of size 31 * 31.

The patch function takes an input image, and starting from the top left corner, it will attempt to create as many patches with the dimensions of the specified width and length with no overlap in order to ensure there is no redundancy in the patches. As a result, if there are pixels that won't fit exactly in the specified patch dimensions, those pixels are discarded. It is possible to start from the opposite side of the image, and generate patches to conserve the discarded pixels, but there would be redundant patches. The redundancy could trigger a bias within classification performance of the FCN and was not pursued.

An interesting dual problem forms: larger patches result in the loss of data, but larger patch sizes also retain the most spatial context. Analysis on this dual problem was performed comparing patch sizes of 8 * 8, 16*16, 32*32, and 64* 64.

Model Architecture

Architecture

The fully convolutional model had four convolutional layers, followed by 3 convolutional layers[2].

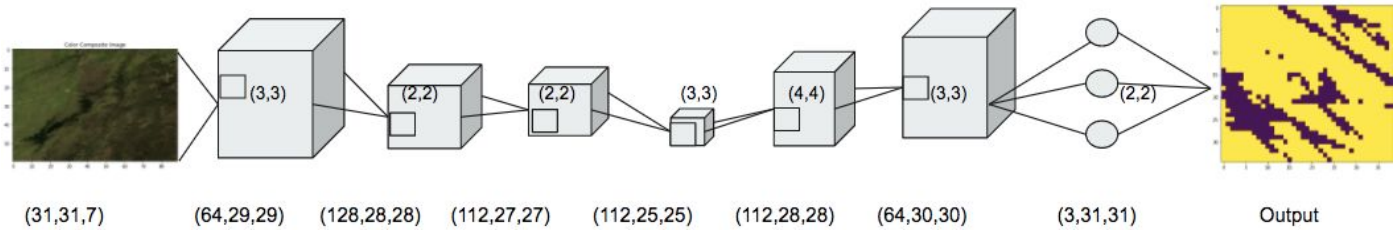

Figure 3 FCN architecture: An example patch size of 31 * 31 was used. Four convolutional layers with hidden units of 64, 128, 112, and 112 were used to learn specific neurons. Filter sizes used were (3,3), (2,2) , (2,2), and (3,3) These were followed by three deconvolutional layers containing 112, 64, and then 3 hidden units respectively to reform data to the original size using filters of size (4,4) , (3,3), (2,2). Each layer was activated using a relu.

After the first convolutional layer, a batch-normalization layer was introduced. After the last deconvolutional layer, a reshape layer was added to get the 4D matrix (N, *w, l, 3*) converted into the 3D label matrix (N, *w*l*, 3) . Softmax was applied to determine the class a pixel belonged to after the reshape. The loss function used was categorical cross entropy.

Why Three Classes?

One important thing to note is the reshape layer reshapes the input to (Number of Patches, *w* * *l*, **3)**. Even though there are only two classes in the dataset (land and water), the model would determine the probability stemming from three classes. The reasoning stems from the original ImageLevel_Dataset contained pixels that were of class “missing”. The model architecture was originally designed with three classes in mind, but upon using the newer version of the data, the “missing” class disappeared.

Furthermore, when the patch generator function shaped the labels to be a binarized class label, when only using two classes the resulting label size created on an **entire split** would be (1, *w* * *l*, 1). This would cause an issue as only one of the labels would be saved for the patch generation for an entire split, resulting in an extremely frustrating bug. For example, out of 6000 training patches, only one patch would have it’s label stored.

An inspection to determine if having an additional class would cause an issue in the classification of a pixel in a non-existent class was performed. It was deduced that the probability of a pixel being of this non-existent class was < 1%, indicating the impact on the classification performance was negligible.

Experimental Results

Patch Dimension Dual Problem

Assigning the same 701 lakes to the training set, 109 lakes to the validation set, and 105 lakes in the test set using 10 epochs, an analysis on the impact of patch dimensions was performed. Each epoch took approximately 60 secs to train with a batch size of 16.

Dimensions	**N Test Patches**	**N Train Patches**	**N Pixels (Test)**	**Land % (Test)**	**Accuracy %**
64 * 64	10	83	40960	99.49	99.49
32 * 32	98	612	100352	95.45	**97.4**
16 * 16	622	3487	159232	90.64	**97.0**
8 * 8	2620	14503	167680	90.47	**95.85**

Table 2 Patch Dimension Statistics

The statistics for the patches looked at the number of training and test patches, as well as the percentage of pixels that were land. This allowed a comparison between a naive classifier and the FCN accuracy. All patch sizes barring 64 * 64 had a classification performance that was better than a naive classifier.

RGB vs 7 Bands Analysis

Using the same training parameters listed above and the same FCN architecture, patches were constructed using only the red, green and blue bands. A quick comparison on the accuracy of the FCN using RGB bands only compared to the FCN using all bands is presented below:

Patch Dimensions	**RGB accuracy (%)**	**All Bands Accuracy (%)**
64 x 64	99.49	99.49
32 x 32	95.53	**97.4**
16 x 16	95.09	**97.0**
8 x 8	**95.90**	95.85

Table 3 RGB vs 7 Band Accuracy

The interesting aspect is that both patches had the same accuracy for 64 * 64 patch size, and RGB performed marginally better for the 8 * 8 patch size. The other patch dimensions favored the 7 band model. A visualization of size 16 * 16 patches is shown below:

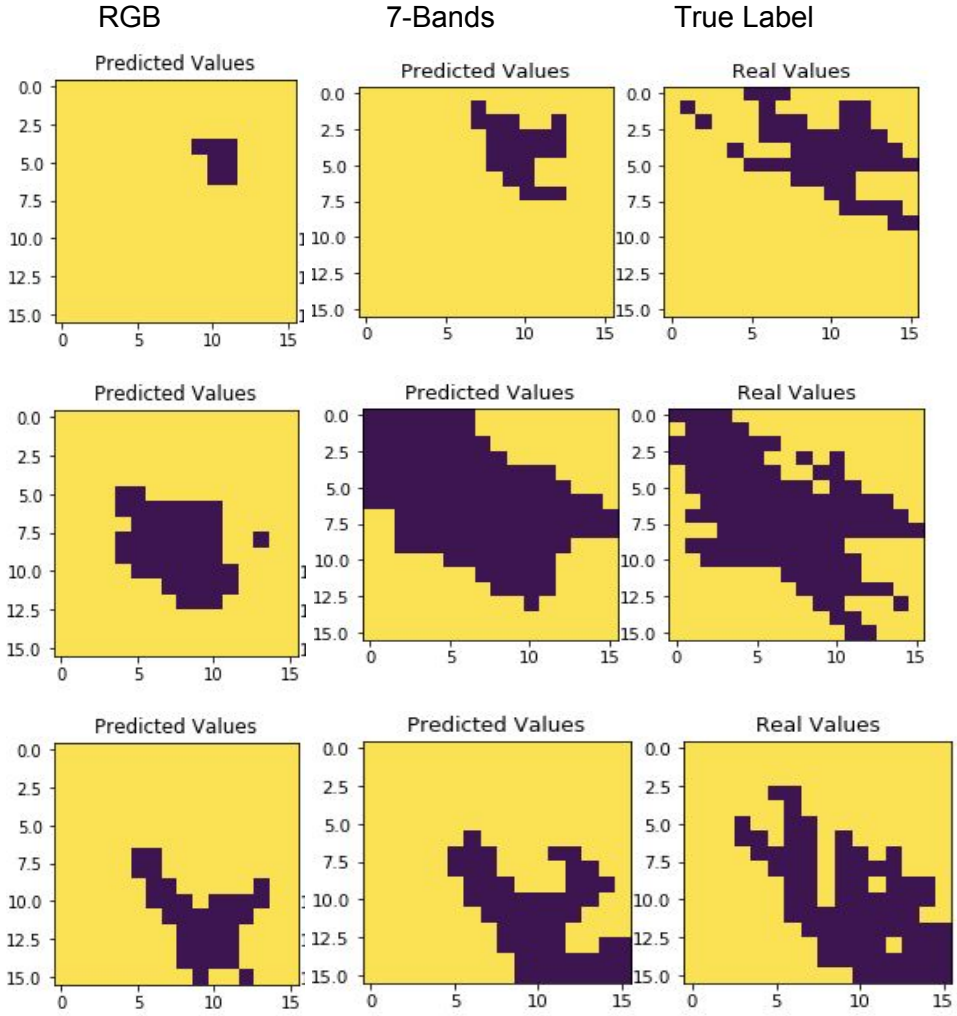

Figure 4 Visualization of RGB and 7 Band FCN performance: Patches were selected at random. Both models show coarse behavior, however RGB tends to be less coarse compared to 7-Bands. Patch size of 16 * 16 was used.

Logistic Regression Comparison

The simple logistic regression was constructed on the PixelWise_Dataset. Ideally, using the corresponding lakes that were used to train FCN was desired, however, the PixelWise_Dataset did not retain the information of its image counterpart. As a result, the training data was randomly split into 1389463 pixels, and was tested upon 463155 pixels.

Class	Precision	Recall	Number of Pixels
Water	0.86	0.64	29220
Land	0.98	0.99	433935

Table 4 Logistic Regression Results

The overall accuracy of the logistic regression was 97.07%, and the percentage of land pixels in the test set is approximately 93.27%. Similar to the FCN, a logistic regression was able to learn a model that performed better than a naive classifier. The logistic regression was much quicker to train compared to the FCN.

Conclusion

Patch Dual Problem

It's evident that patch dimensions play a role in FCN classification accuracy. The first trend noticed is the classification accuracy increased with larger patch sizes. This could be attributed to either a better performance with higher spatial context, or a product of limited testing instances. The biggest difference in the naive classifier and FCN stemmed from using patch sizes of 16 * 16. The underlying reasoning may possibly stem from a three-way optimization: The number of patches, the spatial context conserved, and **the quality of patches**.

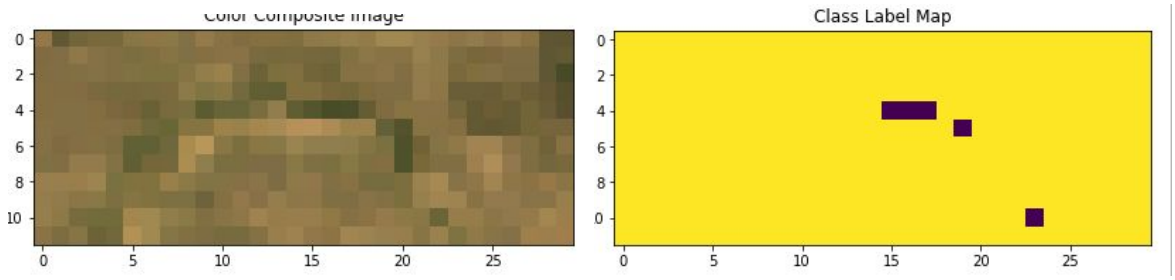

Figure 5: Smallest Image Visualization

An examination of smaller images in the dataset indicates a very coarse image. These images are extremely difficult to correctly classify visually. 16 * 16 patches will include less images of poor quality, which may negatively impact the parameters learned in the FCN. Overall, more images of high quality lakes would be desired to further expand on the analysis.

RGB vs 7 Bands Analysis

The preliminary analysis on the usefulness of the NIR and SWIR bands indicates they may have some merit. The visualization shown indicates that using only the RGB bands for size 16 * 16 patches causes the prediction labels to be more coarse in comparison to using all 7 bands. On the contrary, using all 7 bands seems to generate a prediction more similar in shape compared to the true labels, giving indication that NIR and SWIR bands do have benefits.

The 8 * 8 patches did have a marginally better accuracy using only RGB, however, that may be attributed to the low quality images that were used to train the model. Size 16 * 16 and 32 * 32 tend to have a more significant increase in prediction accuracy, meaning it's also possible that these bands also are more useful as spatial context increases.

Is Spatial Context a Boon?

With the results of the logistic regression conferring a 97.07% accuracy compared to the 16 * 16 FCN's 97.4% accuracy, it brings some interesting considerations if spatial context is useful for pixelwise classification. The first thing to note is the training speeds are much quicker for the logistic regression model. With approximately 60 secs to train one epoch of the FCN, only ten epochs were used. It's most likely that accuracy could be increased as more epochs are run for the FCN. The second consideration is the ability for the logistic regression model to utilize every individual pixel. With reliance on patches, the FCN confers loss of data as not all of the pixels will be able to fit in the patches. This imposes a constraint on images to be predicted to be larger and cleanly divisible by patch dimensions.

The preliminary analysis performed indicates that it's possible to increase the accuracy with the spatial context, but it's unclear the the true extent due to the limited number of patches. FCN's are heavily reliant upon the amount of data that is supplied in the training. With the current dataset, it's unclear if spatial context truly is a boon.

Domain Adaptation

One important consideration is the topic of domain adaptation. With the images stemming from all over the globe at various times, lakes will look different visually, causing a shift in the features values that would classify a pixel.

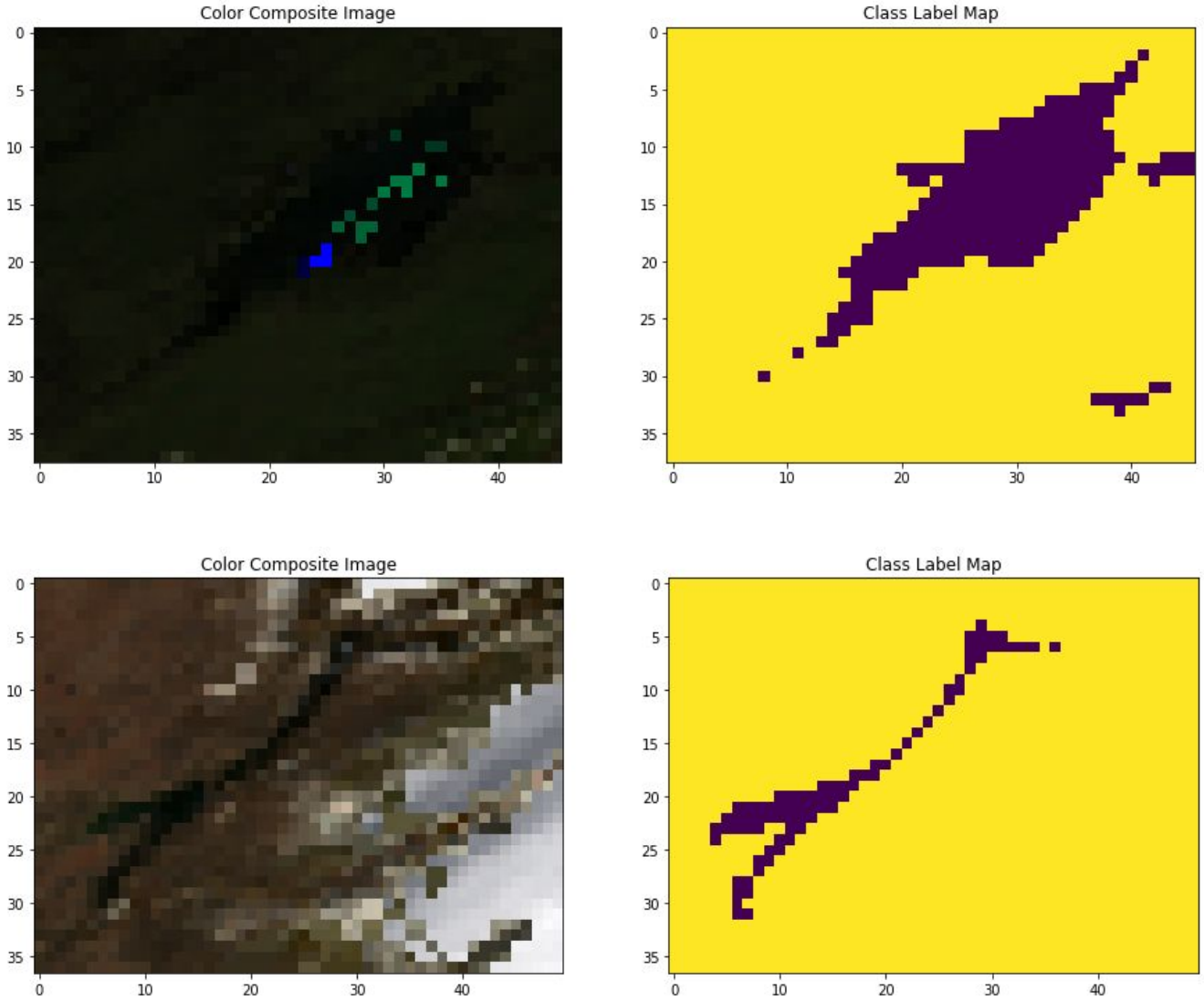

Figure 6 Lakes Present in Different Domains: Two images arep presented using their RGB color spectrum, as well as the class labels. One lake is taken in the night time, while the other lake is taken what appears to be during a snowy weather.

The training set used to learn the features of what constitutes land or lake is highly dependent on how representative the training set is on the test set. Learning the features on lakes in the night time will cause a poor performance on the test set if the lakes are taken in the day. Various methodologies have been proposed for a model to learn the features in a domain invariant region which could be applied to further improve on the FCN's accuracy[3].

FCN Optimization

One important aspect not thoroughly investigated is the optimization of layers, hidden units, and filter sizes of the FCN. Larger patches would allow the ability to utilize kernels with bigger windows, providing features with more spatial context, and less parameters per layer. It's quite possible that each individual patch size has its own optimized machinery which composes of different number of hidden layers, hidden units per layer, and filter sizes. This was kept the same throughout experimentation of patch size dimensions in order to ensure the same machinery could be applied to every patch dimension.

Concluding Remarks

The Fully Convolutional Network performs reasonably well in determining the pixels that are land or water. The FCN relies upon using patches in order to correctly classify pixels. Using four convolutional layers, followed by three deconvolutional layers, the FCN performs better when utilizing all seven bands in comparison to only using the RGB bands. Lastly, a logistic regression model performs comparable to the FCN, indicating spatial context may not be as big of a boon. The most important consideration is that there are limited number of quality lakes in the dataset, and the FCN may prove to be more useful as more images are utilized.

Sources

1). Information about MODIS: https://terra.nasa.gov/about/terra-instruments/modis
2). Paper highlighting FCN: https://people.eecs.berkeley.edu/~jonlong/long_shelhamer_fcn.pdf
3). Domain Adaptation paper: https://arxiv.org/pdf/1702.05464.pdf

Appendix

Github repository (UMN): https://github.umn.edu/bharg016/CSCI-8980-Project

FCN Full Code

```
# importing modules
from scipy.io import matlab
import os
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import urllib.request
get_ipython().magic('matplotlib inline')
import scipy.stats
import tensorflow as ts
import keras
from keras.layers import Dense, Reshape, Flatten,Conv2D, MaxPooling2D,Conv2DTranspose,
Activation, Dropout, BatchNormalization
from keras.models import Sequential
from sklearn import preprocessing


#loading data from the server. This will download the dataset in your current directory
and then load it.
dataset_name = 'ImageLevelDataset_Version2'
url = 'http://umnlcc.cs.umn.edu/WaterDatasets/' + dataset_name + '.zip'
urllib.request.urlretrieve(url,dataset_name + '.zip')
os.system('unzip ' + dataset_name + '.zip')
print('Dataset Loaded ...')

#Get image_names and id names
image_names = os.listdir(dataset_name)
ID_name = []
for image_file in image_names:
    ID_name.append(image_file.replace('data_','').replace('.mat',''))

# Visualize a lake image
ID = ID_name[811]
data = matlab.loadmat(dataset_name + '/data_' + ID +'.mat')
X = data['X'].astype(np.float64)
min_val = 0
max_val = np.amax(X)
X[X[:, :, :] > max_val] = max_val
X[X[:, :, :] < min_val] = min_val
```

```
for b in range(X.shape[2]):
    X[:, :, b] = X[:, :, b] * 2.0 / (max_val - min_val)


plt.figure(figsize = (15,8))
plt.subplot(1,2,1)
plt.imshow(X[:,:,[0, 3, 2]])
plt.title('Color Composite Image')
plt.subplot(1,2,2)
plt.imshow(data['Y'])
plt.title('Class Label Map')
plt.show()


#Function to create model
def model_build(train_X):

    model = Sequential()
    model.add(Conv2D(64, kernel_size=(3, 3),
                     activation='relu',
                     input_shape= train_X.shape[1:]))
    model.add(BatchNormalization(axis = 1, momentum = 0.99, epsilon = .001))
    model.add(Conv2D(128, kernel_size = (2,2), activation = 'relu'))
    model.add(Conv2D(112, kernel_size = (2,2), activation = 'relu'))
    model.add(Conv2D(112, kernel_size = (3,3), activation = 'relu'))
    model.add(Conv2DTranspose(112,(4,4), activation ='relu'))
    model.add(Conv2DTranspose(64,(3,3), activation = 'relu'))
    model.add(Conv2DTranspose(3,(2,2), activation = 'relu'))
    model.add(Reshape(((train_X.shape[1] ** 2),3)))
    model.add(Activation('softmax'))




    model.compile(loss= 'categorical_crossentropy',
                  optimizer= 'adam',
                  metrics=['accuracy'])
    return(model)

#Function to get largest image

def smallest_image(IDs):
    smallest_shape = (10000,10000,7)
    for ID in ID_name:
        data = matlab.loadmat(dataset_name + '/data_' + ID +'.mat')
        if data['X'].shape < smallest_shape:
            smallest_shape = data['X'].shape
            smallest_ID = ID
    return (smallest_ID)


#function to create patches
def Patch_creator(image, X_array, Y_array, horizontal, vertical):

    #Get Features (pixels) and labels for the image
    pixels = image['X'][:,:,(0,3,2)]
    labels = image['Y']
```

```
        #Used to encode
        lb = preprocessing.LabelBinarizer()
        lb.fit([1,2,3])

        #check to see if patch size is bigger than image
        if labels.shape > (horizontal, vertical):

            #Find how many patches can be achieved vertically and horizontally
            Num_Horizontal = math.floor(labels.shape[1] / horizontal)
            Num_Vert = math.floor(labels.shape[0] / vertical)

            #initalize vertical position
            vert_pos1 = 0
            vert_pos2 = vertical -1

            #initalize horizontal position
            horz_pos1 = horizontal * -1
            horz_pos2 = -1

            for vertical_shift in range (Num_Vert):

                #\Get all horizontal shifts for each vertical shift first
                for horizontal_shift in range(Num_Horizontal):

                    horz_pos1 = horz_pos1 + horizontal
                    horz_pos2 = horz_pos2 + horizontal

                    #slices image on dimensions to get patch
                    X_patch = pixels[vert_pos1:vert_pos2,
                                     horz_pos1 :horz_pos2, :].reshape(
                                     1,vertical-1,horizontal-1,pixels.shape[2])

                    #Binzaration and Encode classes (3)
                    Y_patch = labels[vert_pos1:vert_pos2,
                                     horz_pos1:horz_pos2].reshape((vertical -1) *
                                     (horizontal -1))
                    Encoded_Y_patch = lb.transform(Y_patch)
                    Encoded_Y_patch = Encoded_Y_patch.reshape(
                            1,Encoded_Y_patch.shape[0], Encoded_Y_patch.shape[1])

                    #If you have an empty array, create a new one,
                    #Else keep adding patches and info
                    if X_array == []:
                        X_array = np.array(X_patch)
                    elif X_array != []:
                        X_array = np.vstack( (X_array,X_patch))

                    if Y_array == []:
                        Y_array = np.array(Encoded_Y_patch)
                    elif Y_array != []:
                        Y_array = np.vstack( (Y_array,Encoded_Y_patch))

                #Shift vertical Patch
                vert_pos1 = vert_pos1 + vertical
                vert_pos2 = vert_pos2 + vertical

                #reset horizontal position
```

```
            horz_pos1 = (horizontal * -1)
            horz_pos2 = -1

    return (X_array, Y_array)


def Visualize_Patch(Patch_Index,test_Y, predicted_Y, horizontal, vertical):

    #Get the patches
    realpatch = test_Y[Patch_Index,:]
    predictedpatch = predicted_Y[Patch_Index,:]

    #Reshape the patches to have the max probability of the class in the original size
    predictedpatch =
np.asarray(pd.DataFrame(predictedpatch).idxmax(1)).reshape(vertical-1,horizontal-1).ast
ype(int)
    realpatch = np.asarray(pd.DataFrame(realpatch).idxmax(1)
).reshape(vertical-1,horizontal-1).astype(int)

    #plot
    plt.subplot(1,2,1)
    plt.imshow(predictedpatch)
    plt.title('Predicted Values')
    plt.subplot(1,2,2)
    plt.imshow(realpatch)
    plt.title('Real Values')

def get_Land_perc(test_Y):
    land_count = 0
    for patch_num in range(test_Y.shape[0]):
        land_count = land_count +
sum(np.asarray(pd.DataFrame(test_Y[patch_num,:]).idxmax(1)) == 1)

    return(land_count / (test_Y.shape[0] * test_Y.shape[1]))


#Initalize and create Patches
train_X = train_Y = test_Y = test_X = valid_X = valid_Y = []
train_set = ID_name[0:700]
valid_set = ID_name[701:810]
test_set = ID_name[811:955]

vertical = 17
horizontal = 17

for ID in train_set:
    data = matlab.loadmat(dataset_name + '/data_' + ID +'.mat')
    train_X,train_Y = Patch_creator(data, train_X, train_Y, vertical, horizontal )

for ID in valid_set:
    data = matlab.loadmat(dataset_name + '/data_' + ID +'.mat')
    valid_X, valid_Y = Patch_creator(data, valid_X, valid_Y, vertical, horizontal)

for ID in test_set:
    data = matlab.loadmat(dataset_name + '/data_' + ID +'.mat')
    test_X, test_Y = Patch_creator(data,test_X,test_Y,vertical, horizontal)
```

```
#Build Model
model = model_build(train_X)

#Train Model with Validation Set
model.fit(train_X, train_Y,
          batch_size= 16,
          epochs= 10,
          verbose=1,
          validation_data=(valid_X, valid_Y ))

#Evaluate Model
print(model.evaluate(test_X, test_Y))

predicted_values = model.predict(test_X)

Visualize_Patch(0,test_Y,predicted_values, vertical, horizontal)



land_percent = get_Land_perc(test_Y)
print(land_percent)
```