Robin Dehde

Machine Learning Engineer Nanodegree

Dstl Satellite Imagery Feature Detection

I. Definition

Project Overview

The project of choice is the creation of a solution for the Dstl satellite imagery feature detection kaggle challenge. It asks kagglers to analyze satellite images of the earth surface, providing labelled data for classification. Therefore this is a supervised computer vision problem.

The motivation behind this project is the fact that satellite image quantity explodes exponentially in number over time, while usually analysis of pictures has been done by hand or simple algorithms. In the near future this is no longer feasible.

The defence science and technology laboratory (Dstl) invites competitors to create a machine learning approach for their analysis of satellite data of the earth surface. The task that is expected of this software is to scan pictures for structures like buildings, roads and water, and then mark where these structures are in the picture (in the form of multipolygons).

The data that is given already provides 'perfect' multipolygons for every picture. This problem is a little different from typical computer vision problems, as not a whole picture has to be assigned one label, but basically every pixel in every picture has to get assigned a label.

Problem Statement

Kaggle asks competitors to create an application that is able to create labels for each pixel for the satellite images from given polygons, train an algorithm on these labelled pictures/pixels, and make predictions for an unlabelled set of pictures. These predictions then again need to be in the form of polygons.

The satellite images have 19 bands (M-band: 8, SWIR or A-band: 8 and rgb-band: 3). The satellite measured the earth surface at 16 different wavelengths (while an usual camera measures 3 channels: Red, green, blue). The number of different wavelengths is 16 and not 19 as the M-band channel covers the same wavelengths as the rgb or panchromatic channel, plus 5 more wavelengths. The different wavelengths have different resolutions. 3 of the bands are the usual rgb pictures at high resolution, 8 cover the wavelengths from 450-100nm (called M-band) and further 8 cover the near

infrared from 1000-2000nm (SWIR or A-band) . The rgb pictures are given at a resolution of roughly 3300x3300 pixels while the M-band has 830x830 and the A-channel is provided with a resolution of 130x130 pixels. It will be part of the problem to preprocess the different parts of the data in a way that makes them manageable for an algorithm.

The polygons will be transformed into binary pixel masks and back using the python library shapely. Furthermore for transforming binary pixel masks back to polygons it is first necessary to find contours in the pixel masks in order to be usable by the shapely package. Therefore contours are found in the masks using openCV 2. This returns a mask that only contains the contours. For every class that is to be predicted there will be one mask.

The satellite images are in the tiff format and will be read into numpy arrays with help of the tifffile package. In this format they can also be enlarged, shrinked, normalized (with zero mean and standard deviation one for each channel) and further preprocessed (cropping the images, augmenting the data by flipping and rotating it).

At first it will be very interesting to see how often each of the 10 classes appears in the trainset, and how the ratio is between classes. This might be very important to reach a balance between classes, in order to give each class the same weight on the learning process.

Only now one can start to experiment with the actual machine learning part of this challenge. Since there are many wavelengths given in the image, it makes sense to first explore how good the different classes can be distinguished just by purely looking at the values of the different wavelengths (not yet looking at structures, like a human would naturally do). Since there are many different wavelengths, it is hard to visualize every class for every channel. Also it is to be expected that rather combinations of different channels may be good predictors for certain classes. To try and find out how much information is already coded in the different channels, the first apprach will be pixel based only. Since the trainset is larger than the RAM of my system, many sklearn algorithms fall away, and this will be done using the SGDclassifier, since it supports out-of-core computation.
This learner will be used to create pixel based predictions, getting all wavelengths as input and outputting a probability whether this pixel truly belongs to a certain class. This probability will then be thresholded to produce a binary mask that can be turned into polygons.

The next step then is to incorporate structural information. This means that for classifying a certain pixel many pixel surrounding it are used for the prediction as well, thereby hopefully being able to capture the structure of trees, streets, buildings, waterways and so on. This will be done using a convolutional neural net, as CNNs are optimized for working well with pictures. The CNN will be implemented using google's tensorflow library. It will take in cropped parts of the picture and return labels for every pixel.

Metrics

Since this is a kaggle challenge there is a straightforward metric: The metric that is used to compare competitors. In this case this is the jaccard index of the submitted polygons for the unlabeled satellite data. The jaccard index between two areas measures what amount of area are shared, compared to how big each area is (the

intersection of two areas divided by their union). The position in the kaggle leaderbord will inform about what performance maximally to expect and what performance should be quickly available.

The jaccard index is a very reasonable metric to use in this competition. Too many false positives as well as false negatives lead to an decreased score. The complete leaderboard score is the average over the 10 jaccard indexes, one for each class. A perfect prediction would result in a value of 1.0, while a prediction that did not match a single real polygon would result in an index of 0.

Furthermore, the first pixel based prediction will result in a certain submission score. Any attempts at also incorporating the structural information of surrounding pixels should be able to outperform the pixel based approach if done right.

Also the jaccard index can be calculated on the training set to realize performance while training, best done as cross validation.

II. Analysis

Data Exploration

The data is given as images in .tiff format. One image corresponds to an earth surface area of a squared km. Pictures of these surfaces are given as:

~3300 x 3300 x 3 (RGB)

~830 x 830 x 8 (multispectral band or m-band)

~130 x 130 x 8 (SWIR or a-band)

The different channels cover the following wavelengths:

- Sensor : WorldView 3
- Wavebands :
    - Panchromatic: 450-800 nm (rgb)
    - 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

There are 25 labelled images provided. The labels are provided as polygons. In order to make a submission to kaggle, competitors need to create labels in the form of polygons for 429 satellite images. For these 429 images no labels are provided. The submission has to contain the multipolygon data for each class for each image.

As RGB pictures (panchromatic) provide the highest resolution, they will probably interesting for classes that are very small (cars, motorcycles, trucks, single trees). Though the size of this data could be too much for the limited system configuration that is my laptop.

The m-band will most likely yield good detection rates for most objects, as it contains a reasonable resolution, mixed with a wide scope of wavelengths covered. Not only is the scope wide, but the visual spectrum tends to have the bigger differences in absorption/ reflection for different objects and different wavelengths, compared to their neighbours in the spectrum (namely infrared and UV). That is to say, not much variance is expected for a certain pixel for various channels of the SWIR channel, generally speaking.

Lastly the SWIR channel has probably a too low resolution to be of help for many small classes. It could rather be interesting for detecting big classes, such as waterways and lakes or buildings and wide streets. As infrared tends to contain temperature information about the object, relative temperatures might be detectable (water warmer or colder than the surrounding terrain, etc).

For the m-band channel, the train set consists of 17,347,200 pixels (834*832*(25pictures)). Of these pixels, 4.27% make up buildings (class 1). The other percentages follow in the table below:

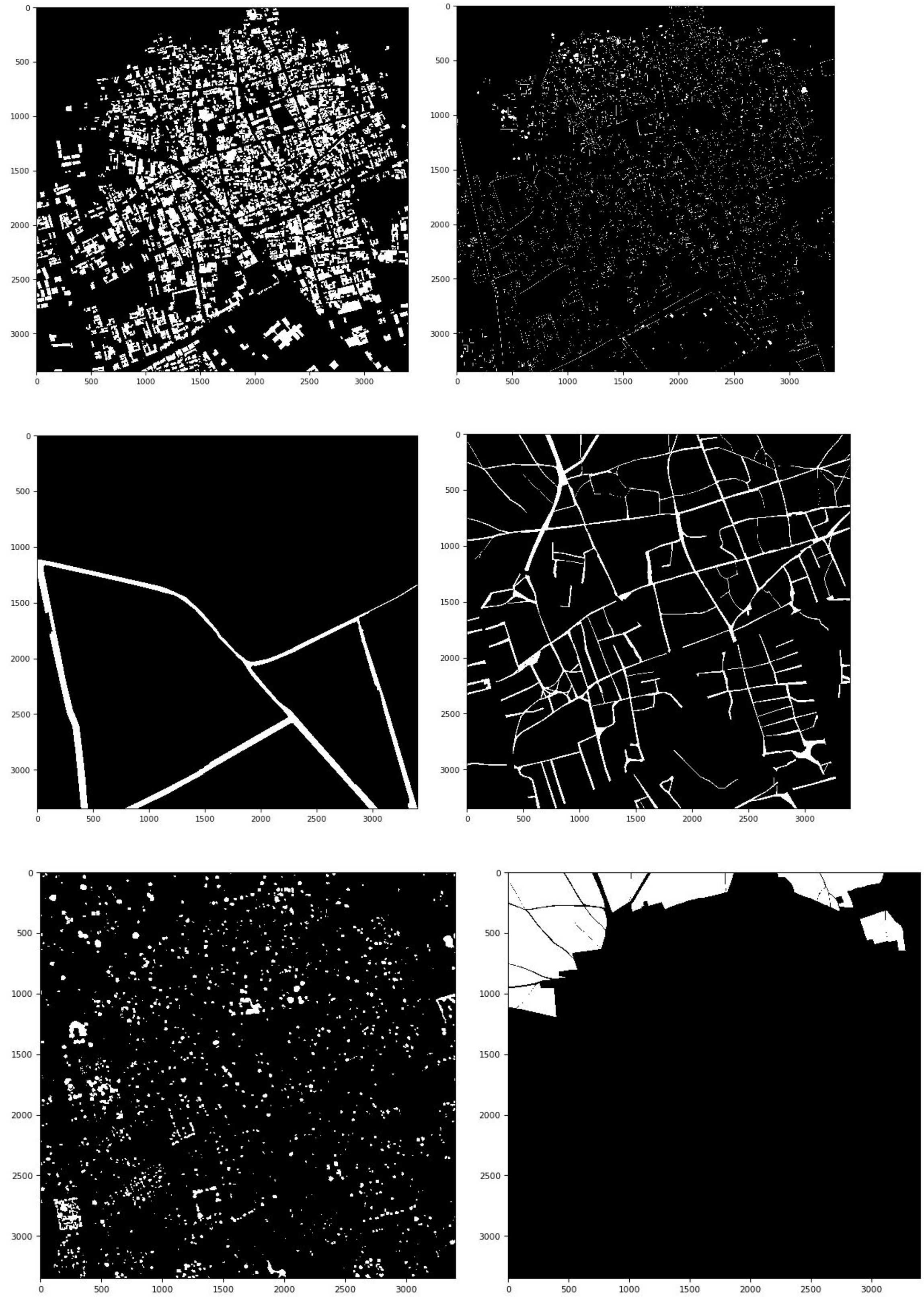
class	1	2	3	4	5	6	7	8	9	10
%	4.27	1.60	0.89	3.15	13.23	27.89	0.53	0.19	0.006	0.04

Therefore, the classes appear highly unequally. This is a problem, especially when training a single classifier on all the classes. The highest difference between class appearance is on the order of magnitude 4 (crops to small vehicles, class 6 to class 9). The most frequent class appears in 4,838,134 pixels, while the least frequent class appears in 1040 pixels. This can be addressed by augmenting the classes that do not appear often enough until all classes appear roughly equally often. This augmentation could be added noise, rotation by angles, cropping, as well as just showing the classifier certain crops or pixels more often than others.

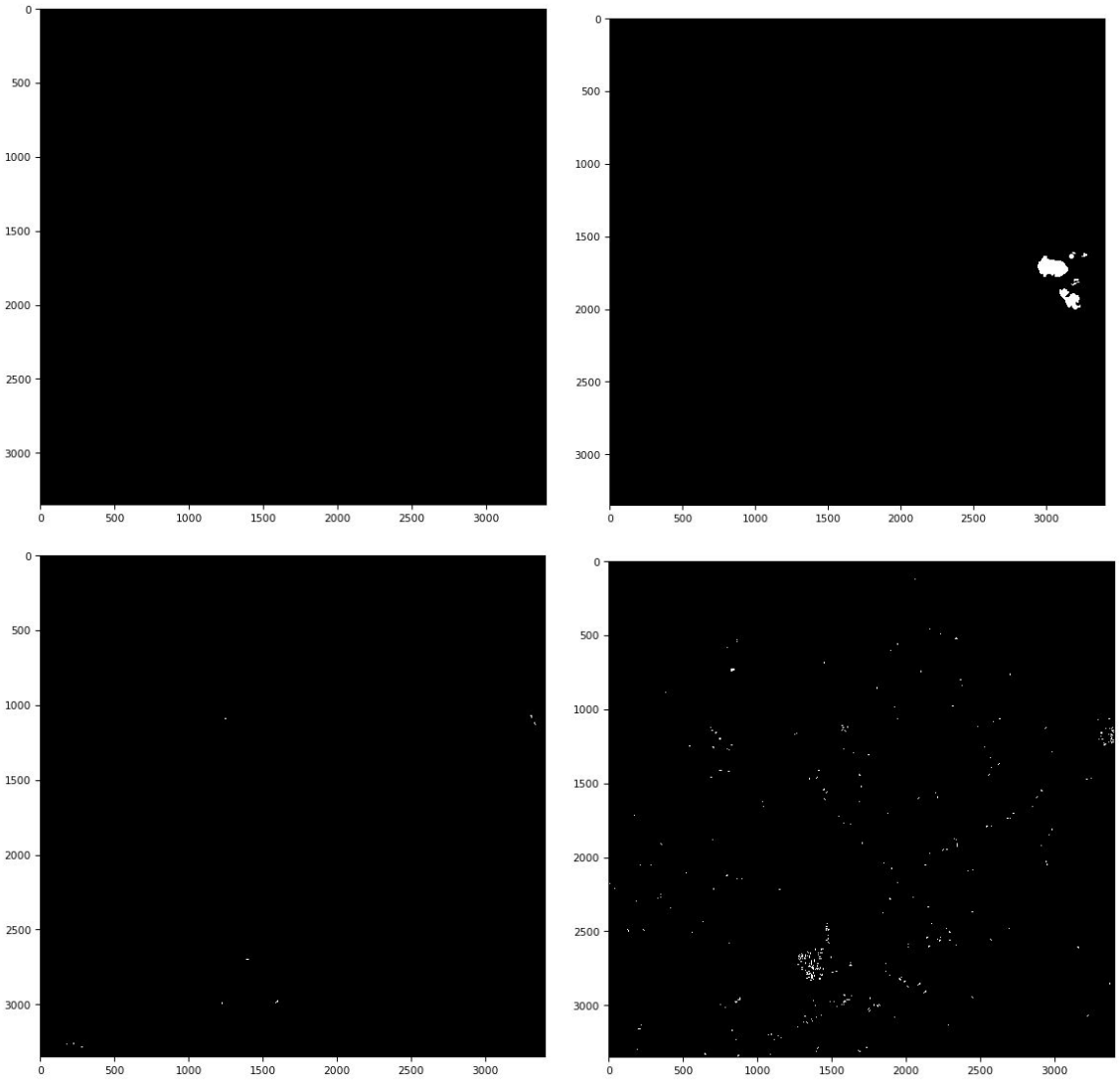
Lastly, the picture size differs from picture to picture. Therefore means will have to be taken to either make the algorithm take any shape or beforehand resize the images that the algorithm takes as input.

Exploratory Visualization

The most meaningful thing to take a look at in the beginning is obviously the ready to be interpreted by the brain rgb images of the satellite data. Also it might be a good idea to plot the masks for the different classes as well in order to get an idea of what each class looks like in satellite pictures. The classes are plotted from 1 to 10, therefore containing: Buildings(1), Miscellaneous Manmade Structures(2), Streets(3), Track(4), Trees(5), Crops(6), Waterway(7), Standing Water(8), Large Vehicles(9), Small Vehicles(10)



0 500 1000 1500 2000 2500 3000
0 500 1000 1500 2000 2500 3000



This visualization was chosen as it gives a nice introduction into the data, as well as stress complications with the dataset that were mentioned before.

With nice introduction to the data I mostly mean that with a short peak at the picture, it is rather clear, that a pixel classifier should be able to capture at least some classes. Here especially buildings seem to have the same colour. Yet on the other hand not much success is to be expected from doing this for streets and roads, as they appear to have the exact same colour but only vary in their respective widths.

Furthermore, as mentioned before, the extreme inequality in how often each class appears definitely needs to be addressed.

Looking further into the shapes of the masks for the classes there seem to be many patterns. Misc. manmade structures together with cars seem to have the finest resolution and are found near buildings. The decision between street and tracks will probably be problematic, as to the human eye not much difference is visible here. Trees can be seen in the pictures rather well, while crops appear to have the same dusty colour as the other surroundings of this small village. Lastly the standing water

is hard to see. It also appears in a rather dusty colour.

The colours of the above RGB picture are a bit stretched for better contrast, as is probably immediately clear when looking at it (at least not many blue roofs where I live). Therefore the dusty colour is perhaps not exact, but what counts is that from only looking at colours it will probably be hard to have good results. But nevertheless it should be possible for a human to roughly find all 10 classes correctly in this picture after some training. Therefore this could be possible for an ML algorithm as well, given the right information.

Algorithms and Techniques

The algorithms that will be used in this project are the SGDclassifier from scikit-learn for pixel classification and convolutional neural nets in the tensorflow library for area based classification of each pixel.

SGDclassifier stands for stochastic gradient descent classifier. This algorithm was mainly chosen as it allows classification for out of core computation problems, as the trainset does not fit in my RAM in its entirety, and throwing away parts of the dataset did not seem reasonable. Now from the choices that sklearn offers for out of core computation the SGDclassifier was particularly interesting, as a simple and small change in initialization of the loss function changes the underlying algorithm, thereby providing an easy and fast possibility to vary algorithms.

The SGDclassifier implements regularized linear models with stochastic gradient descent learning. This means that each input (in this case the colour channels) are each multiplied by a weight and then summed together with a bias (weight multiplied by 1). This creates a value that can be positive or negative. Usually this can be directly used to assign a class: a negative value means this pixel does not belong to the class in question, while positive means that it does belong to the class. The learning is done by creating a loss function that is minimized via stochastic gradient descent. This descent tries to minimize the difference between the prediction of classes and the actual class, while by the way also trying to keep the weights small altogether by applying regularization. This regularization can be chosen to be l2 or l1 (the value of the weights squared or just the magnitude of the weights). Learning steps created this way are multiplied by a learning rate that is decreasing over time. This has the effect, that later samples do not undo the learning of samples in the beginning, or at least making this harder. This can also have the effect that yet unseen but important features are neglected when they happen to accumulate at the end of the learning process. Therefore data shuffling is, as in most of the cases, very important.

SGD requires scaled mean and variance in order to work properly. Therefore it is practical to use sklearn’s standardscaler, which analyzes every picture and scales the values accordingly. Here all the images were first analyzed by the standardscaler and then these results were fit to every picture.

The standard variables of the SGDclassifier that concern the algorithm itself are: loss='hinge', penalty='l2', alpha=0.0001, l1_ratio=0.15, n_iter=5, shuffle=True, learning_rate='optimal', eta0=0.0, power_t=0.5. Their meanings are respectively:

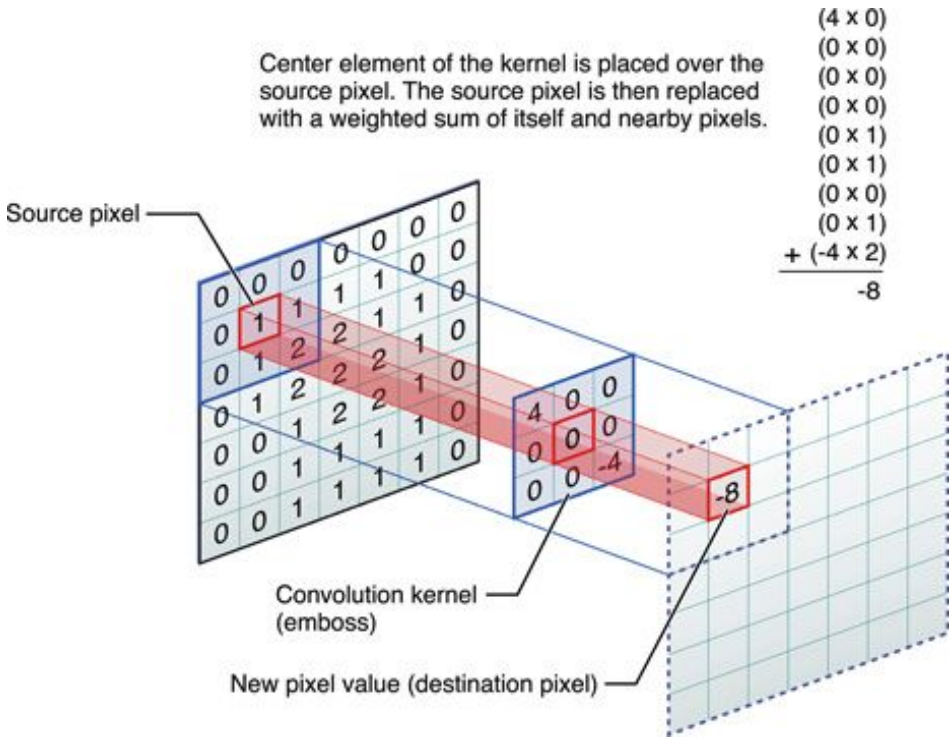
The hinge loss basically results in a support vector machine algorithm that learns with

stochastic gradient descent. The l2 penalty adds an additional term to the loss function that is the square of every weight, or the weight vector squared. The alpha hyperparameter denotes the relative weight between the regular loss and the l2 penalty. It is multiplied on the l2 regularization. The standard value of 0.0001 makes sure that the l2 regularization is only a secondary influence on the loss minimization, therefore only kicking in when weights get extremely large. The l1_ratio is only used when the elastic net is used as penalty. This will not be used here. 'N_iter' describes how often the algorithm will go over the the entire dataset or the partial dataset when using 'partial_fit'. The shuffle keyword shuffles the train data before training on it. This does not work though on the partial_fit, though it is not explicitly stated in the documentation. It is therefore important to note that shuffling needs to be performed before training. The learning rate determines the function with which the algorithm's learning rate will decay. Optimal results in a decay as: eta = 1.0 / (alpha * (t + t0)), where t0 is a heuristic based on the size of the dataset. 'eta0' is only used for a constant learning rate instead of the optimal learning rate. In this case the learning rate does not change over time. 'power_t' is only used when using the 'invscale' keyword for the learning_rate. This then results in an exponential decay of the learning as: eta = eta0 / pow(t, power_t).

The second algorithm that is used are convolutional neural nets. Neural nets basically rely on a Perceptron as the primary building block. These perceptrons are just combined in certain ways to create certain architectures. It is very important that the activation function of the perceptron is a nonlinear function as otherwise the hidden layers in a CNN will not improve the algorithm (as a mixture of linear functions will just result in a linear function).

A perceptron gets certain inputs, for example the 16 channels of the satellite data for one pixel. Every channel has a certain weight assigned to it. Every input value is now multiplied by its respective weight plus another bias weight. The resulting number is then fed into the 'activation function'. This is typically a sigmoid, but in general this will definitely be a nonlinear function in deep neural nets (as otherwise adding further layers will have no effect). Therefore, any input is mapped onto an interval between -1 and 1 by every 'neuron' in a neural net. In neural nets, this output will then be part of the input of the next layer. Also layers can have very different widths (perceptrons/neurons per layer).

Now this general structure is still very flexible, as now you can create any architecture you want starting from the primary building block of the Perceptron. Many papers have been written on different architectures and techniques used in combinations. What will be used here will be a rather simple neural net, as computational power is a big issue in this challenge. The structure will contain at least one convolution, as convolutions are a good way to process images in machine learning algorithms, since this process captures the geometry of the pictures it gets as input. As convolutions are a very important concept of neural nets, deep learning and picture processing in general, much can be said about convolutions. I will outline the basic principle here.



The picture above shows the process of a convolution. The ‘source pixel[s]’ would be for example in the challenge the value of the red channel of the rgb-picture. Now in the first layer of the neural net, there is a convolutional layer. In the picture above, a so called convolution kernel of 3x3 is used (the 3x3 box of numbers in the middle). This is mathematically speaking the convolution function with which the original input is convolved. Now in a convolutional layer, every neuron basically tries to learn such a convolution kernel. The size of 3x3 is given before, but the actual weights in the kernel (here 4,0,0,0,0,0,0,0,-4) are learned by the neuron step by step. Therefore in a convolutional layer every neuron will try to learn a filter function, that will then create a new picture of the old one, that is hopefully as useful as possible for the layer behind this convolutional layer in order to determine the class this picture belongs to. Therefore the number of neurons in a convolutional layer corresponds to the number of different filters that will or can be learned for the input of that layer. A further important hyperparameter of convolutions is the padding. Two kinds of padding are supported in tensorflow. In the picture above, when you move the filter one step to the side and so on, you create a new picture that is denoted in the last plane in the picture. As can already be seen, if you only use the numbers of the first plane, the last plane will hold less numbers than the first (The first plane has dimension 7x7, the last plane will have dimension 6x6 this way). Using only valid numbers that are actually part of the input is referred to as ‘valid’ padding. The other tensorflow supported padding possibility is ‘same’ padding. It is called same, since around the input picture there will be as many zeros added where needed around the picture to get the resulting picture (the last plane in the picture above) to have the same dimensions as the input. In the example above this would be a ring of zeros around the picture, making the input picture have the dimension 8x8 and therefore result in a 7x7 output picture after filtering. The last hyperparameter for convolutions is the stride for the convolution. This means whether or not the convolution kernel will skip pixels in the input picture. In the picture above a stride of 1 would mean that the 3x3 box will move one pixel to the right in the next step. A stride of 2 would mean that the 3x3 box

moves 2 pixels to the right, etc. This again results in smaller end pictures.

Further pooling layers (typically max-pooling) will then make the algorithm invariant against the exact position of a certain feature in the data it gets (whether a tree is to left or the right might not matter, while it might matter that there is a tree nearby). This process also has a certain kernel size, let's say 2x2. This kernel will then as above go over the entire picture, using a certain stride as before. Out of these 2x2 pixels, the max-pooling layer will take the largest value and only return this value, thereby reducing the 2x2 input into an 1x1 output. Pooling is often used to reduce the dimensionality of the input for further layers as well.

The neural net typically uses cross entropy as the loss function, in order to maximize the confidence with which classifications are done. Here it makes sense though to directly implement the Jaccard index as the loss function, as it is the metric that counts in the end. Furthermore it has been shown, that a l2 regularization is equivalent to augmenting data by putting noise over images. Therefore l2 regularization was chosen to be implemented.

Neural nets are also trained by using the stochastic gradient descent method, minimizing the respective loss function. Tensoflow also offers statistically more advanced and enhanced version of the SGD. The one that performed best in this case is the ADAM optimizer. Much can be said about how to enhance optimizer, but this would go far beyond the scope of this report. It should suffice to say that ADAM computes adaptive learning rates for each parameter and additionally stores an exponentially decaying average of past squared gradients. This leads to faster convergence as this optimizer is less prone to be bothered by 'walls' in the loss function that keep it from walking towards the minimum, as well as being less likely to be drawn into local minima.

Benchmark

Two benchmarks will be used in this report. The first is the end result of the pixel-wise classification of the SGDclassifier class. The last version of the SGDclassifier had a kaggle public leaderboard score of 0.10142. This gives an estimate what is minimally expected of a neural net that is used on the dataset. If this is not reached, something is goign very wrong in the implementation.

The second benchmark is the best public leaderboard score of the jaccard index, which is: 0.58585. This basically marks what is possible when using good techniques, a well adapted architecture but also and very importantly: Extreme computational resources that are not available to me. Winning teams posted that they bought additional graphics cards for this challenge as training already took several days and still took extremely long. This is obviously not feasible in the scope of this project, yet it is interesting to see what is possible with a neural net solution.

III. Methodology

Data Preprocessing

Quite a few non-trivial preprocessing issues had to be overcome for this challenge. The csv file of the training polygons has to be transformed to a format which is usable by the algorithms. Different channel picture sizes had to be transformed to be able to be used as input together. Then output masks have to be transformed back into polygon data that is then again stored in a csv file. Furthermore both SGDclassifier and neural nets both profited from standard-scaling the input channels. As one neural net was used for the different classes, it was also important to make the different classes appear uniformly throughout the training set in order to represent each class equally strong in the learning process. All these preprocessing steps will be addressed in the following.

1. **Join channels, create one size for all pictures, cut picture into pieces**
   As each channel bundle (referred to as band) has a different resolution, each band is given as a separate .tiff file by kaggle. It is convenient to join the different bands into one numpy.array. In the same step it is useful to transform every .tiff file into the same resolution, as the different files tend to slightly vary in the exact number of pixels. A look at the M-band data shows that the smallest resolution is 834x832. This is used in the following as the standard resolution. For the neural net the A-band is upscaled to this resolution. First the pictures are read into numpy.array using the tifffile package. This is followed by using the cv2.resize() function with cv2.INTER_CUBIC for upscaling (p-band) and cv2.INTER_LINEAR for downscaling (M-band images that have a higher resolution than the smallest picture).

   Furthermore the resulting arrays are cut into 10x10 pieces. This is done to make sure that the neural net gets fed different pictures in every batch it receives later in the process. The cutting is done using appropriate looping and indexing of the arrays. Resulting arrays are saved using the numpy.savez function.

   For the neural net the resized pictures are here combined with the training masks in the same size. Turning train polygons into masks is handled in point 2.

   This procedure is not used for the pixel classifier, as it was not necessary. The pixel classifier only gets the M-band channel and does not care about picture size, as each pixel is analyzed independently.

   The panchromatic band is thrown away here, as the high resolution results in too high computational demands. This should be kept in mind though, since the high resolution might offer good help for determining the small classes such as small vehicles and large vehicles.

2. **csv -> polygons -> pixel masks -> ML -> pixel masks -> polygons -> csv**
   At first, the training images need to be identified, as train and test set are delivered together. Therefore the csv list train_wkt_v4.csv is read and image ids are stored in a list. When training, the images are loaded one by one using these image ids. Now for every picture there is a loop over all 10 classes. For every class and picture the corresponding polygons are loaded from train_wkt_v4.csv' via the

function shapely.wkt.loads() from the shapely python package. The picture is read into a numpy array using the tiff.imread function of the tifffile python package. The polygons now need to be scaled to the pixel-size of the picture. The exact way of scaling is given by kaggle:
https://www.kaggle.com/c/dstl-satellite-imagery-feature-detection/details/data-processing-tutorial. Basically the polygons are stretched in a certain way to match the picture size.
In the function get_scalers these stretching factors are determined and then applied using the shapely.affinity.scale() function.

Now a training mask can be created from the scaled polygons. This is done in the mask_for_polygons function. In it an empty numpy array is initialized. Then the exterior structure coordinated of the polygons are written into a list called exteriors. Then the cv2 function cv2.fillPoly is used to write 1's into the numpy.array where the exterior coordinates say there is a polygon present. This is only the rough structure so far. The inner structure is found by looping over the poly.interiors and writing the coordinates into a list. This list is then again given to cv2.fillPoly which will then overwrite 1's from above with 0's where the inner structure says that there is no polygon at this precise coordinate. The numpy.array is returned.

Both the features and the labels are given as numpy.arrays of the same size and machine learning can be performed here. The ML algorithm will return a numpy.array. For the training of the algorithm, preprocessing would be done here. In case of the validation dataset, the predictions of the algorithm need to be transformed back into polygons and written in the correct order into a csv file. Creating polygons from array pixel masks is handled in the mask_to_polygons function. With the help of openCV this function finds contours in the array (cv2.findContours). These contours are then approximated (cv2.approxPolyDP). This means that finer structure of polygons is neglected. How much is neglected is determined by the parameter epsilon. A higher value means less details, thereby decreasing the amount of information necessary. This is a parameter that can be tuned later for each class. Next the different contours that were found before need to be joined to a set, using the hierarchy that the cv2.findContours also returned. Now some filtering is done for too small contours that contain very few pixels and are probably just noise. For achieving this the cv2.contourArea function is used to compare the size to another parameter called min_area. A higher min_area throws away more polygons, allowing only bigger polygons. This is interesting especially for classes like waterway and crops which tend to be huge polygons. Next the arrays that were manipulated as described are turned into multipolygons using the MultiPolygon class of shapely.geometry. In this process it happens that invalid polygons are created. These need to be cleaned out, using the MultiPolygon.buffer() function. As this sometimes turns multipolygons into polygons, the polygons will be turned into multipolygons if their type is polygon. Now these polygons are returned by the function.
These polygons now still need to be transformed back to a normalized size for submission. This is again done by using the shapely.affinity.scale() function with the scalers from above. This prediction is then put into the wkt format by using the function shapely.wkt.dumps.
Now these polygons need only to be written into a csv file in the right order (using appropriate looping for each class for each image).

3. Standardscaler for SGDclassifier

The SGDclassifier needs normalized input in order to function well. Therefore in the process above the pictures that were read into numpy.arrays via the tifffile package

need to be normalized. This can be easily done using the StandardScaler() class from scikit-learn. For this the StandardScaler.partial_fit() function is applied to every image available (validation as well as training set). This finds the mean and standard deviation for every wavelength channel. This is then applied to every picture-array just before the ML part, using the StandardScaler.transform() function. This then subtracts the mean and scales values in the channels to be given as factors of standard deviation.

4. **Min/Max normalization for CNN**
   For the CNN the input of each channel is normalized by subtracting the minimal value for that channel for all train images and then dividing the result by (max - min). This results in a values that lie between 0...1. This turned out to benefit the learning of the CNN. This is achieved by looping over all channels and using the numpy.max and numpy.min functions of the train data and then applying the necessary mathematical operations.

5. **Data augmentation for CNN**
   The 25 train pictures that were cut into 100 pieces each are loaded spontaneously by the function make_clts_data. In it random numbers are drawn to determine whether a picture piece is rotated and/or flipped. Flipping and rotation is achieved with the numpy package. This is done to create more training samples and make the CNN invariant under rotation of the input. Rotations are only available for 90/180/270 degrees or no rotation.

6. **Uniform class representation for CNN**
   As some classes are very weakly represented in the data, it is important to balance this ratio for training one neural net on all classes. This is achieved by drawing samples for each class separately. This will result though in the algorithm analyzing many times the same samples for an underrepresented class. This leads to overfitting on these samples on a test set, or a bias towards the real dataset. Still it is important that each class appears equally often for the algorithm to learn the underlying concept of each class, at least at first. In order to then also learn how often each class really appears, this uniformness of classes is slowly deactivated over time. Therefore the learning process starts with equal amounts of samples for each class, but with every training step this uniformness moves further towards the real rate of which classes actually naturally appear in the dataset. This showed small improvements compared to no uniformness. But in the end it might make most sense to train 10 CNNs, 1 for each class. This would terminate the fight of different classes over who changes the weight, but at the cost that cross correlations between classes are lost and mutually beneficial structure-concepts are not learned together.

Implementation and Refinement

As I find it extremely hard to differentiate implementation from refinement, I couple these two sections in the following. The process of implementation happened very iteratively and strongly coupled with refinement.

The first goal for this challenge was to create a pixel classifier that takes the M-band (UV up to infrared in 8 channels) data as input, yielding a number between 0 and 1 as output for a certain class. Given from the data preprocessing is the input data in the form of numpy.arrays.
The first naive attempt was to build a SVM that trains on the combined trainset of 25 pictures. This turned out to be too big for the RAM on my machine though. The solution I came up with was to use the out of core computation algorithms provided by

scikit-learn (though later on I updated my RAM as this was becoming a serious problem in every way that needed way too much attention for small gains). Therefore the SGDclassifier crystallized as a good choice. The only other real competitors was the multinomial naive bayes. Since all that was wanted from this first classification was to get an estimate how much performance is easily available from pixel classification, the generalized linear models used by the SGDclassifier seemed very reasonable.

Now the goal was to get the SGDclassifier working using the partial_fit(features, labels) function. This was at first naively implemented, looping over every image and every class, using one SGDclassifier per class. This was before the uniformness efforts were made, therefore it was clear that one classifier per class was necessary at this point, due to the high class inequality. The naive implementation had really bad results though. This was due to the problem that no shuffling was used, but one picture after another was presented to the algorithm. Therefore the algorithm learned the first picture by heart, then the second and while learning the second by heart already forget everything it learned about the first picture. This problem was addressed by reading in a certain amount of percent of each picture at a time (~5% *25 pictures) and then shuffling these training points. This improved performance drastically and basically resulted in the first score for this challenge: 0.09 as jaccard index.

The performance estimates for the above were done by calculating the jaccard index, precision and recall. Precision and recall are not really important here but were just used to get familiar what the jaccard index intuitively describes in terms of the known metrics of precision and recall. Therefore the only real metric used is the jaccard index. As the SGDclassifier was used to give out numbers between 0...1 for every pixel and every class, but polygons allow only for a yes/no classification, the output probability masks needed to be turned into binary masks. This was done using a certain threshold (in the beginning 0.35 for all classes). Any number bigger than that was rounded to 1, any below became a zero. From the train mask and this binary mask the true positives, false positives and false negatives were calculated to calculate precision, recall and jaccard index. Furthermore, it was clear that the preprocessing steps of creating polygons from the binary mask would also lose performance. Therefore it was also interesting to see how much performance would be lost in this process. To find this out, polygons were created from the binary masks and then compared with the train polygons (using: final_polygons.intersection(train_polygons).area and .union). This delivered the jaccard index that would be calculated in the end by kaggle. This was mostly worse than the binary masks, but for some classes even better (probably due to eradicated noise with the min_area).

In writing this down this seems almost trivial but was actually a HUGE amount of work. I never worked with openCV or polygons before, so getting used to this and understanding what was going on was a big hassle. Also finding out that the 'shuffle=True' does not work in the SGDclassifier when using the partial_fit function took quite some time to find out (this is not to be found in the documentation…). This was then done by hand.

So far no test set was used. Instead submissions were created and validation set scores were calculated directly by kaggle.
Now follows the part that I actually wanted to do: Getting started with neural nets. I have never worked with neural nets before, though I did attend a machine learning lecture that covered deep learning theoretically. Also I am aware what each part of the

algorithm should theoretically accomplish.

The first idea was to get a typical standard architecture for picture processing working. There was a kaggle kernel on a UNET architecture within keras that I thought was worthy to play around with to get started with neural nets. This did not work on my machine though since there was not enough RAM available for the train set to be loaded at once. This was the moment I bought an extra RAM for my laptop. Then loading the files became possible but this still took excruciatingly long. Therefore I decided to use a much smaller net architecture that my laptop could actually handle and installed the GPU supported tensorflow library to hopefully speed up the process.

In order to get started with tensorflow I did the MNIST data tutorial. There a really small net is used to get rather good first results on the data. As here the computation time was reasonable (training in a matter of hours) I decided to start with a rather shallow net, of up to at most 2-3 layers, and not in the UNET architecture as this already cost too many resources for the MNIST data. It was also decided to train all classes in one net in order to hopefully minimize the training time as similar structures would be learned together across classes (computation time was really a major issue at this point).

In order to be able to use the neural net that was created for the MNIST data, the data now needed further preprocessing, as described in point 1 under preprocessing. From the beginning the 16 channels of M- and A-band were used (UV-infrared at middle resolution and pure infrared at low resolution).

The exact setup of this neural net will be discussed in the following. The first setup was just a single convolutional layer of 100 neurons that is again convolutionally connected to the 10 output neurons (10 for the 10 classes). The last connection is usually fully connected, but was explicitly chosen to be done convolutionally. For the input of the roughly 80x80 pixel stitches this results in predictions for every pixel. A full connection to the predicting 10 neurons would result in a single label for the 80x80 stitch, which could also be done, but would have to be used in a different way. Here basically several 80x80 stitches are used as a single batch for learning. The first covolution had a convolution kernel of 10x10 using the padding 'SAME'. This of course makes the predictions near the border less reliable, as for a border point up to three quarters of the input are zeros. This inspired the last neural net setup that will be discussed later, where data is processed live.

All weights were initialized normally distributed via tensorflow.truncated_normal(). The training happenend with an implementation of the jaccard index as loss function. Furthermore the loss was l2 regularized (which is mathematically equivalent to augmenting data with noise).

At this point the train set was split into train and test set. The neural net never saw certain stitches of the train set, that were then used for calculating a score during training. To see the performance for different classes, not only the overall jaccard was implemented, but also the individual jaccard for each class. Within the jaccard index calculation some constant terms had to be added in order to avoid division by zero errors. The learning was done using the AdamOptimizer from tensorflow, as it performed faster und reached better results compared compared to the regular gradient descent.
Submissions were then made using one whole validation set picture at a time. For scaling the input with the min and max of the respective channels, the same values were used as for the training. The transformation from probability mask to binary to

polygon was the same as before. The first net of only one convolutional layer (i.e. one hidden layer, counting input layer and output layer) with 100 neurons reached the kaggle validation result of 0.04348, which was worse than the pixel classifier. But it was clear that the general structure of the neural net was probably capable of much better results, since this setup was just too simple and was never even capable of overfitting (this was read out of the training behaviour. Performance steadily improved on the test set up to a certain value. The performance never got worse but also could not improve further starting at a certain threshold. This was interpreted as a sign that bias was still the dominant source for bad results.).
Therefore a second hidden layer was put in place. Since calculations were done on the graphics card and the VRAM was strongly limited, the size of the net was also rather limited. The result was a net with a first hidden layer of 60 neurons and convolution kernel of 17x17. The second hidden convolutional layer had 100 neurons. The second layer is convolutional, but in effect is a fully connected layer (again, this sort of convolution is used to create a prediction for every pixel). After some hours of training until no further improvement was visible (still no sign of overfitting) a kaggle validation score of 0.19157 was reached, more than double the score of the naive pixel classifier. A little optimization of the threshold, epsilon and min_area values for creating polygons and a training session of 24 hours increased this score up to 0.21834.

Now the question became how to further improve the neural net without consuming more computational ressources, or even less ressources. Going back to CPU would yield the capacity of using more layers and neurons, but would take longer by a factor of roughly 100 for the exact same net (learning on the order of hours -> order of days!). The last big issue that needed to be addressed and would most likely not consume any further ressources was the uniformity of classes.

As mentioned in the preprocessing part, this was achieved by drawing the classes separately. This implementation demanded bigger structural changes and therefore it was a good time to implement a data wrapper class and da neural net class to also clean up the code. The cleaner code was created with help of 2 friends who have way more experience in object oriented coding, which was a good training (therefore we also joined teams in kaggle at this time). The data wrapper has a function 'getMiniBatch' in which a specified number of items per batch are asked for. The class knows what the frequency of each class is in the training set. Yet the rate which will be used to draw samples from the data of each class is determined by the factor (1./self.class_rate)**(1.-self.uniformness). The self.uniformness starts out at 0.99 and slowly decreases later on in learning. This results in classes being nearly equally represented in the beginning and then being represented as they truly are in the train data at the end.

Though I was sceptical at first whether this could improve final results, it luckily worked. During the training process the jaccard index is overestimated on the training set. Towards the end of training (again roughly 24h) the train set jaccard was around 35%. The kaggle validation score was 0.29999 for this setup with optimized threshold, epsilon and min_area.

In order to optimize epsilon, min_area and the thresholds for each class, the prediction masks for the trainset created by a neural net were used. From these binary masks, polygons were created using a grid of different parameter combinations. The different values that were used are:

threshold	.15	.2	.25	.3	.35	.4	.45
epsilon	.5	1.	3.				
min_area	1.	3.	5.	10.	15.		

These 'optimal' parameters were determined by using the parameters that reached the highest jaccard index. A further submission was done using these parameters, yet for some reason the result was lower than for an educated best guess. The educated guess that was used was eps=2, min_area=4., threshold=0.25 for all classes and resulted in a validation score of 0.29999, while the optimized parameters resulted in a validation score of 0.29582, therefore marginally worse. This is probably due to differences between the train set and the validation set. Therefore the optimization of the parameters suffers from overfitting on the train data. Also a big problem with the optimized parameters was to get a stable solution. Certain combinations of parameters led to the opencv function becoming unstable and creating unusable polygons that had to be fixed by hand to avoid errors in submission. This was done for one submission and after that declared neither worth the time nor the effort, considering the 80/20 principle. Yet the optimization was useful to reach the educated guess.

IV. Results

Model Evaluation and Validation

The final model is a convolutional neural net with 2 hidden layers with 100 neurons in the 1st hidden layer, 60 hidden neurons in the second layer and 10 neurons in the output layer. The data is processed live to reduce the strain on computational ressources (cropping to 1st convolution kernel size, random rotations, flipping). Weights are initialized normally distributed. Learning is done with the Adam optimizer and a learning rate of 0.005. The net used 16 of the 19 available frequency channels, as the rgb does not contain any new spectral information, but only detailed spatial information that is not usable by the used computer setup due to the high strain on computational ressources.

The complete derivation is described under 'Implementation and Refinement'. In short the reasons that made this model setup the winner are:

- Strain on computational resources allows only use of small architecture and 16 channels
- Adam optimizer accelerates learning process
- The biggest possible architecture was chosen, as there was no sign of overfitting
- Above an convolution kernel of 17 again computational strain set in. Though it was tested whether a smaller amount of neurons and bigger kernel would reach better results. This was not the case. The number of neurons is already very small (compared to winning architectures for this challenge).

The model makes use of spatial information (convolutions) as well of the spectral information (16 channels) and therefore makes sense, thinking about the initial expectations about the challenge.

The model's capacity to generalize to unseen data was tested with a validation dataset (the kaggle-check). Now, after the competition not only the public scores are available (test on 1% of the 349 images) but also the private score. The public score for the final model was 0.29999. The private score was 0.23486. It is worth mentioning that for all submissions the score drastically lowered. It is rather probable that in the submission set many unseen features were present, especially when you compare the difference in size of training to validation set (25 compared to 349 images). Therefore I would argue, that the model generalizes well.

As was realized only after the competition, there were a few more perturbations present in the data, that could be further addressed with preprocessing (though I was not even aware that these things could be meaningfully processed). For one thing the training labels seem to be translated relative to their actual counterpart in the pictures in some training pictures. This is discussed here:
https://www.kaggle.com/visoft/dstl-satellite-imagery-feature-detection/correct-image-missalignment-v2/code
Furthermore all pictures are relatively shifted to each other in their 'whiteness', as can be seen here:
https://www.kaggle.com/davidthaler/dstl-satellite-imagery-feature-detection/a-relatively-quick-look-at-the-big-picture
In the link above, all 25 pictures are plotted next to each other. It is clearly visible that the overall picture whiteness changes from picture to picture. This is also basically another added noise source on the set.
What I want to argue with both these cases is the robustness of the presented solution. Even though at least two more sources of random noise across images and within images were present, the model could generalize well enough and find underlying concepts that were still able to generalize to 349 unseen pictures rather well.

It is an interesting question whether the model can be trusted. This trust question probably has to be split across the classes. The model reaches relatively good conclusions for crops and trees and can draw preliminary maps for these classes. These are also the classes that are best represented in the data. For the other classes the algorithm should definitely not be trusted. This model should so far be probably mostly be viewed as a starting point that shows that neural nets might very well be capable to achieve good results, so that there might be enough information in the data to reach conclusions about the 10 classes. The model should definitely not yet be used as a sole classifier for satellite data. To reach that far, more training data, a complexer architecture and highly improved computational resources are necessary.

Justification

The final solution ranks 139 out of 419 on the kaggle leaderbord, making it a solution in the top 34% (my kaggle profile: https://www.kaggle.com/robindehde). The private leaderboard score was 0.23486. The highest achieved private leaderboard score was 0.49272, roughly double my score. In the top ten the lowest score is 0.42669, the top 50 reached 0.29378. The median result on the leaderboard is 0.09055 for the public

and 0.11491 for the private leaderboard. Interestingly, low results had a tendency to improve from public to private leaderboard. This is roughly the result of the pixel classifier that was built here. Though there is much space to the top, I do not think much more can be reached given the computational limitations.

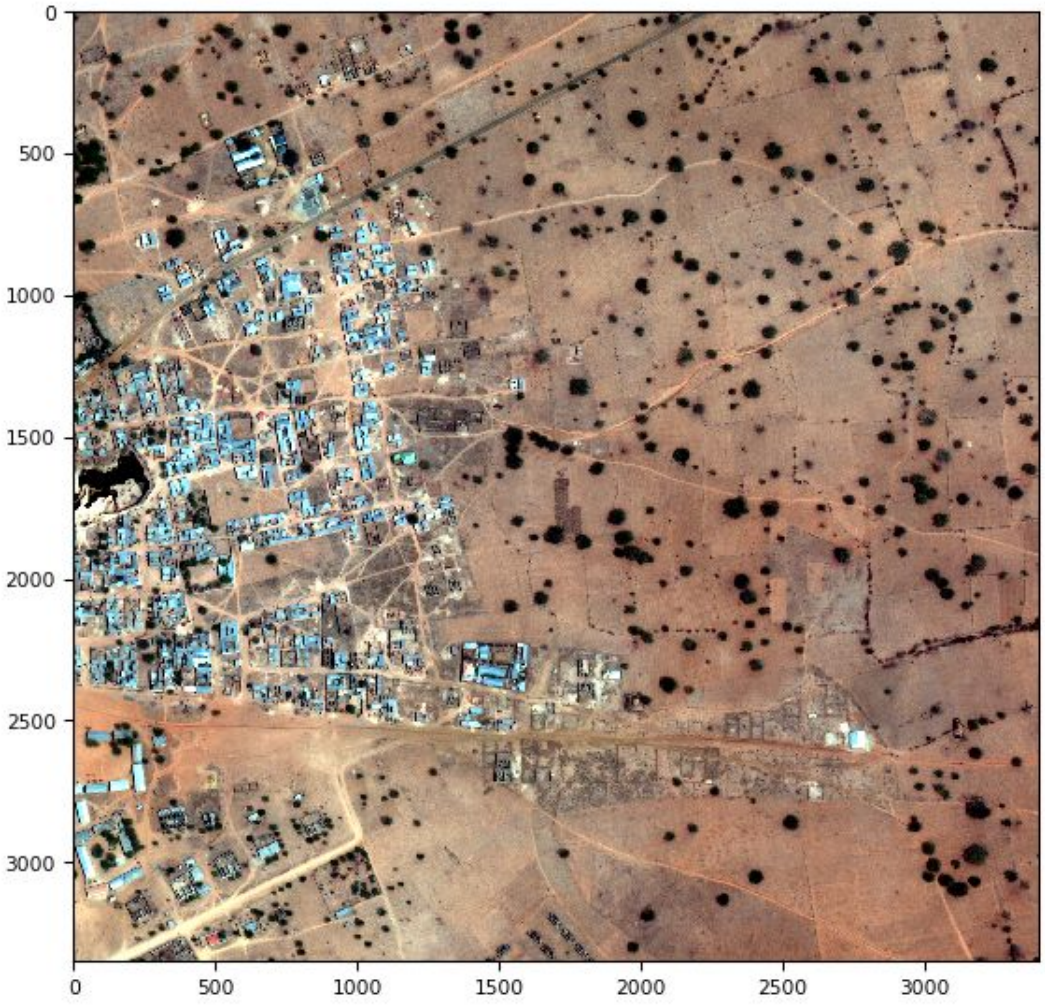
The benchmark of the pixel classifier reached a public leaderboard score of 0.10142 and a private leaderboard score of 0.07821. Therefore the neural net performs roughly three times as good a a generalized linear pixel classifier. It is by the way interesting to note that basically every submission performance dropped by 20% from public to private leaderboard.

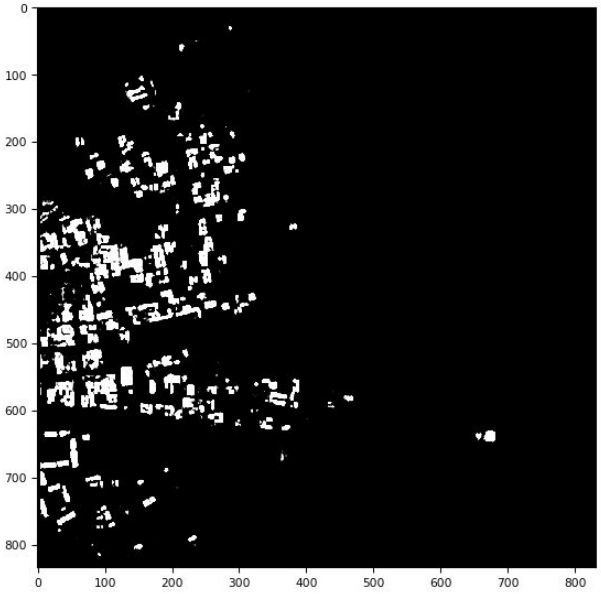
The final solution is not significant enough to have solved the problem. And I would argue that neither is the solution of the 1st place. A Jaccard index of ~0.5 still misclassifies roughly 50% of the time. This can at least not be used for all classes, each class would have to be viewed separately. I am quite certain though that 25 images can never be enough to learn all complexities in satellite data of the whole earth surface. Much more training data is necessary for that. Though I do not think that even the DSTL thought that was possible, but rather the 25 images were chosen to see which approach leads the furthest. If this is the true goal of this challenge, then I am quite certain the final model presented here is capable of showing that neural nets are a very promising solution to the posed problem.

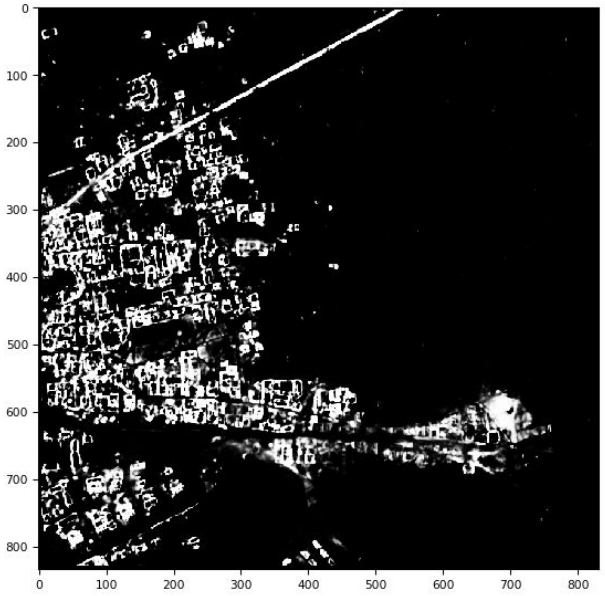
V. Conclusion

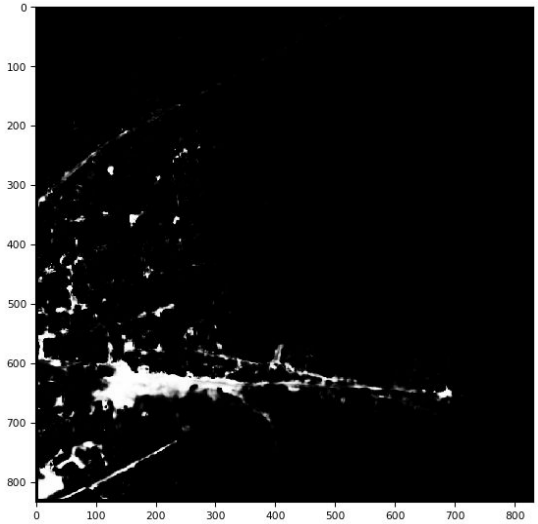
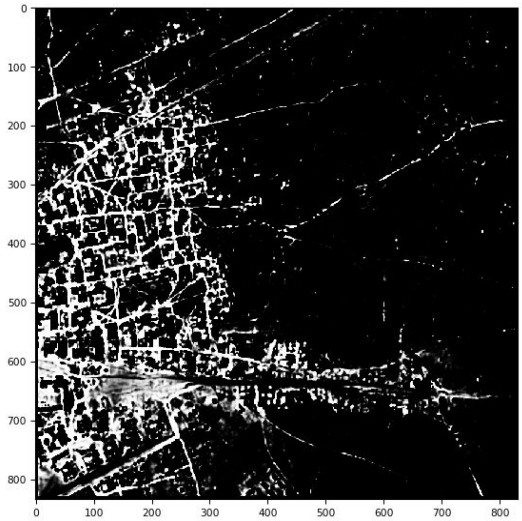
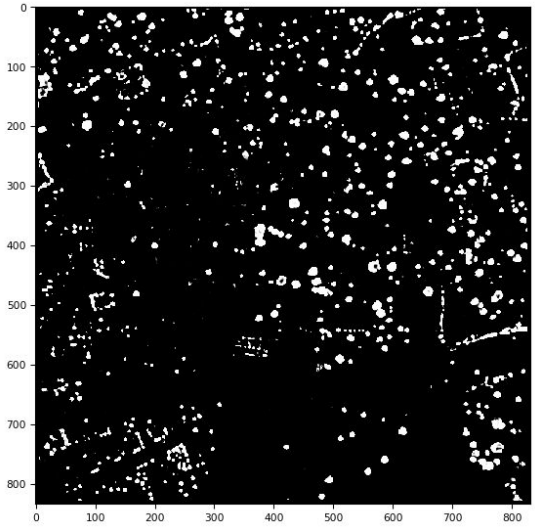
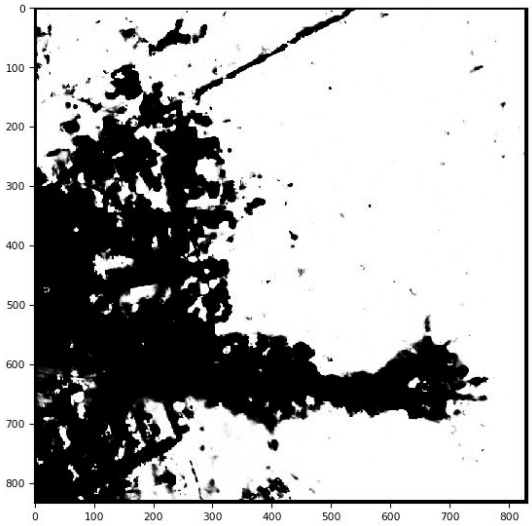
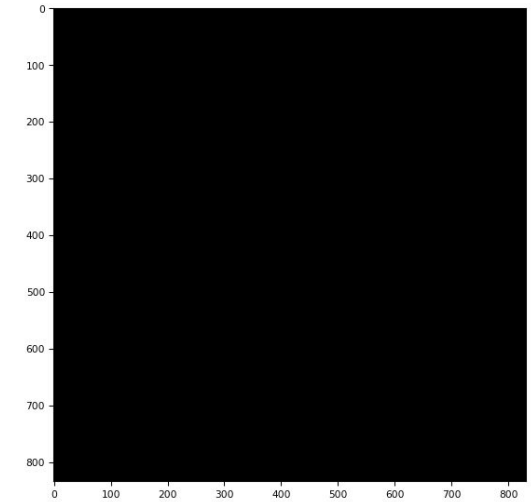
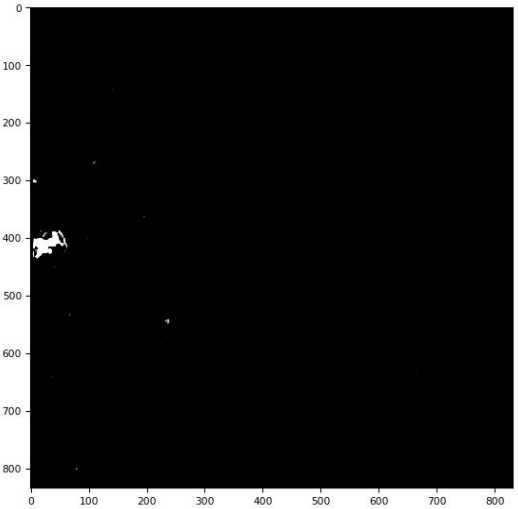
Free-Form Visualization

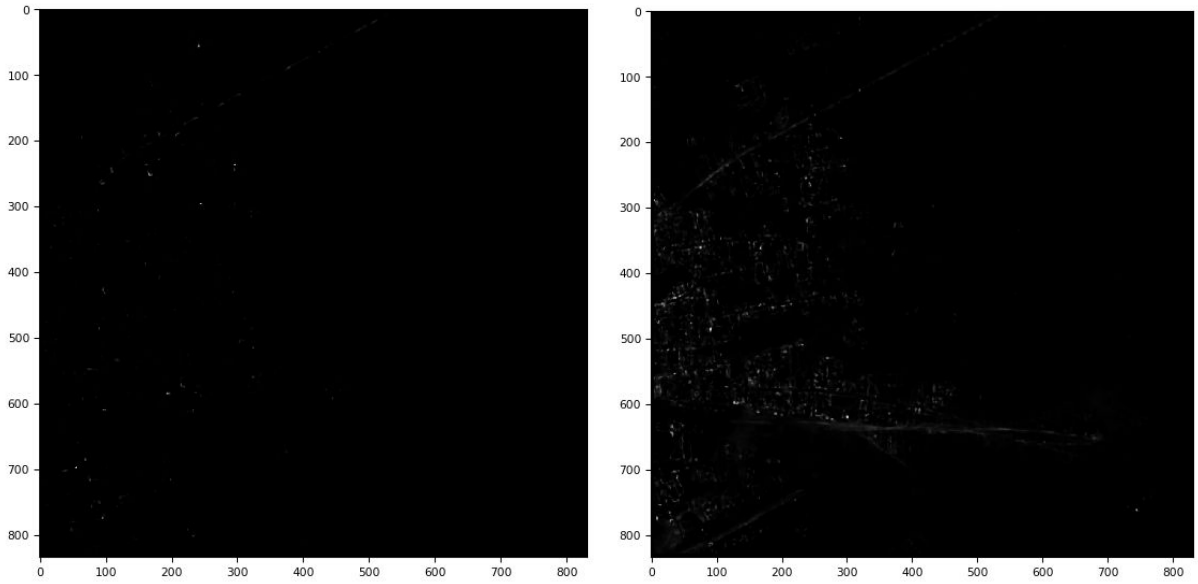
To conclude this project I would like to present one of the kaggle submission pictures together with the masks that the neural net produced for this picture. The neural net has never seen this picture before but is still able to provide rough estimates for the different classes. The colourful picture will be the real picture in rgb, while the 10 following black and white masks correspond to the 10 classes in their order (buildings, manmade structures, streets, tracks, trees, crops, waterway, standing water, vehicles large, vehicles small). White stands for the class being present at this pixel as predicted by the neural net. The labels on the axis of the rgb picture correspond to positions, which is irrelevant for this visualization. Only the structures seen in the images are of relevance.


0
500
1000
1500
2000
2500
3000


0
100
200
300
400
500
600
700
800


0
100
200
300
400
500
600
700
800



These pictures show that the neural net seems to perform well on structures that are also rather easy to see for the human eye. For example buildings, trees and crops are very well visible and seem to be rather accurately represented by the prediction masks. Tracks also seem to be rather well detected, while the algorithm apparently can not make much sense of the concept of streets. The algorithm correctly saw the small lake to the left and correctly predicts no waterway in this picture. For cars it becomes extremely difficult to tell where cars are supposed to be present with the human eye on this small scale. Though the results for the vehicles classes are rather certainly not significant, judging from the weak performance even on the train set.

Overall this visualization shall present one part of this challenge that made this project so interesting: You could basically watch an algorithm learning how to draw maps that one could use in real life, by just giving the algorithm satellite data.

Reflection

The 'Dstl Satellite Imagery Feature Detection' kaggle challenge certainly proved to be challenging. In this project satellite data of the earth's surface was classified into 10 categories. Training labels that were provided in the format of polygons were transformed to binary training masks. 16 channel .tiff pictures were stitched together to gain a multispectral view on the earth surface. Images were preprocessed channel wise to enable proper function of the machine learning algorithms. With this supervised learning dataset both a pixel wise support vector machine classifier for each class and an area based neural net for all classes were trained. The neural nets results were improved by making class appearance uniform. Results were analyzed and especially the neural net architecture improved. A validation dataset was analyzed by both algorithms and predictions (transformed back to polygons) submitted to kaggle in order to determine the algorithms' performance. Parameters for polygon creation were tuned.

This project turned out to not only be much fun but also extremely demanding. The design of this challenge left extremely many ways to solve the challenge. In the end both pixel classifiers (more advanced than mine, found on kaggle discussions for this

challenge) and neural nets turned out to perform well. The polygon creation routine gave rise to additional parameters to be tuned, but also made it possible to have some sort area based approach, even if just a pixel classifier is trained, as in the end such parameters as min_area will decide whether points stay or not.

What turned out to be extremely frustrating about this challenge was the demand on computational resources that were by far not met by my computer. This led to many necessary workarounds. I would have loved to actually try out high performing neural net architectures. Every winner of this competition described how they bought additional graphics cards for this competition. This made the challenge a bit ungrateful, as it seems that the score does not reflect the amount of work that went into this project.

Improvement

A possibility for improvement that immediately comes to mind is solving the most frustrating problem in this challenge: Computational resources. Buying computation time or buying better hardware and leveraging this to create deeper and more advanced architectures (good candidate would be the U-Net architecture).

The higher computational power would also enable using the rgb channel resolution. This might be very useful for detecting very small classes like small vehicles that might just be not well enough resolved in the m-band. Also there are possibilities like panchromatic sharpening, where the m-band datas' resolution can be enhanced using the rgb data. This would hopefully unleash the whole potential of this dataset.

This is also true for the pixel classifier. Having enough resources to get all of the training data into the RAM enables many more possibilities. A very promising thing might also be to investigate known absorption coefficients of certain structures. These coefficients exist for example for standing water, waterways, streets, tracks and many more. These were successfully leveraged by a kaggle competitor to achieve a score of 0.45 in the public leaderboard, using a pixel wise classifier.