INDIAN INSTITUTE OF ENGINEERING SCIENCE & TECHNOLOGY

MINIPROJECT FOR THE SESSION 2020-21

IMAGE-SEGMENTATION

A PYTHON PROJECT FOR SEGMENTATION OF SATELLITE IMAGES

our work on github

Our Team

NIRABHRA MAKHAL

510519043

PARNABRITA MONDAL

510519022

NIKHIL JOSHI

510519110

RAJDEEP GHOSH

510519002

CHIROJIT MANDAL

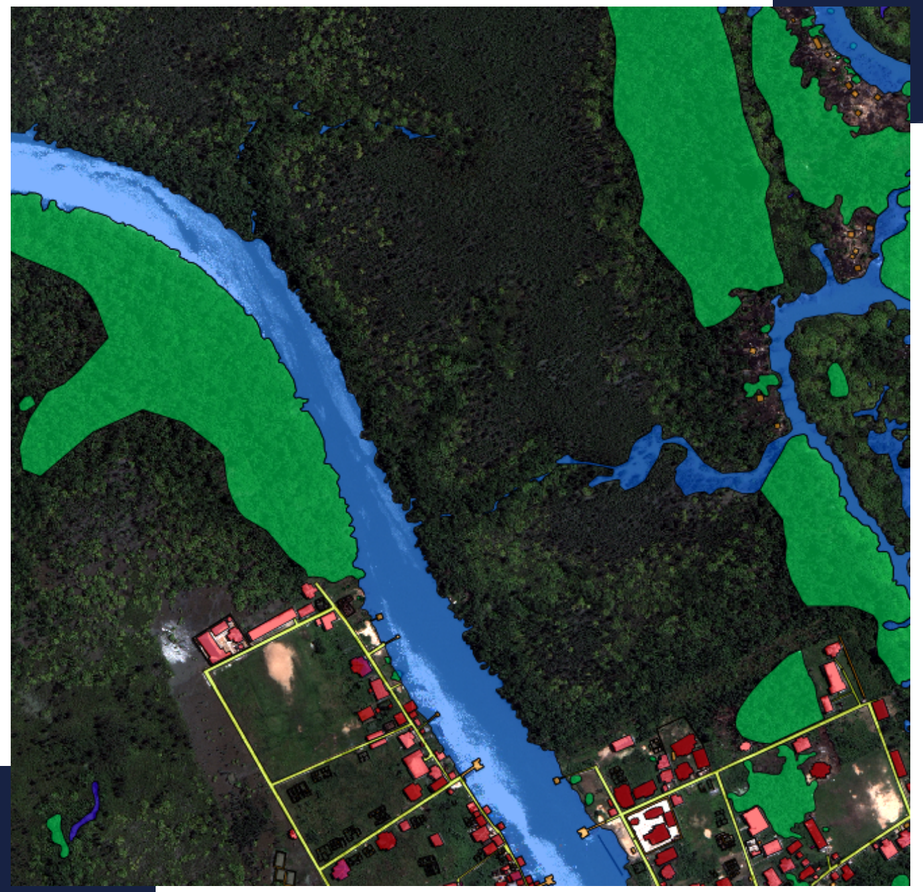
510519032

DHRITIN DUTTA

510519021

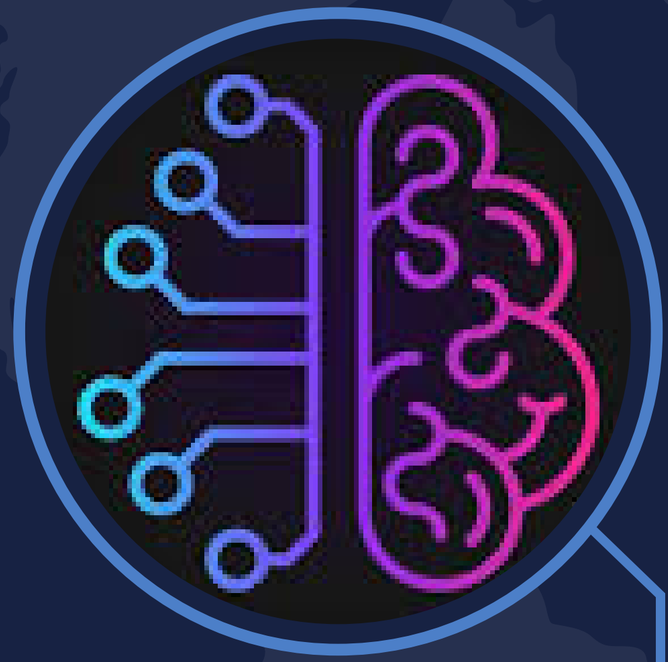
OBJECTIVE

Python code for image segmentation



STEP:

IMAGE MASKING



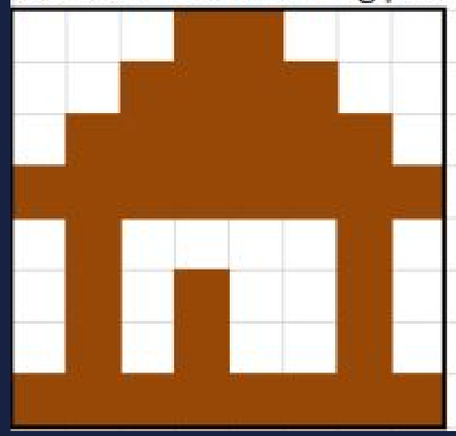
ABOUT THE MODEL

A brief introduction to Run-Length encoding

Run-length encoding (RLE) is a form of lossless data compression in which runs of data (sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run. RLE is a simple yet efficient format for storing binary masks. RLE first divides a vector (or vectorized image) into a series of piecewise constant regions and then for each piece simply stores the length of that piece.

For example, given M=[0 0 1 1 1 0 1] the RLE counts would be [2 3 1 1(Corresponding to the 2 zeroes, 3 ones, 1 zero and 1 one)

Consider the following picture with brown pixels(b) on white background(w)



w	w	w	b	b	w	w	w	3w2b3w
w	w	b	b	b	b	w	w	2w4b2w
w	b	b	b	b	b	b	w	1w6b1w
b	b	b	b	b	b	b	b	8b
w	b	w	w	w	w	b	w	1w1b4w1b1w
w	b	w	b	w	w	b	w	1w1b1w1b2w1b1w
w	b	w	b	w	w	b	w	1w1b1w1b2w1b1w
b	b	b	b	b	b	b	b	8b

OUR MODEL & CODE

GENERATION OF THE MASK

The image is passed as a numpy array, mask and the function *runline_encoding* returns a runline encoding of the mask.

```python
def runline_encoding(mask):
    flat_mask = mask.flatten()
    padded_mask = np.concatenate([[0], flat_mask, [0]])
    run_arr = np.where(padded_mask[1:] != padded_mask[:-1])[0]
    run_arr += 1
    run_arr[1::2] -= run_arr[0::2]
    encoding = ' '.join(str(run) for run in run_arr)
    return encoding
```

GENERATION OF THE MASK

The *generate_mask* function accepts the runline encoding and a dictionary consisting of the labels(taken from the MS coco dataset) and returns a one-dimensional numpy array.

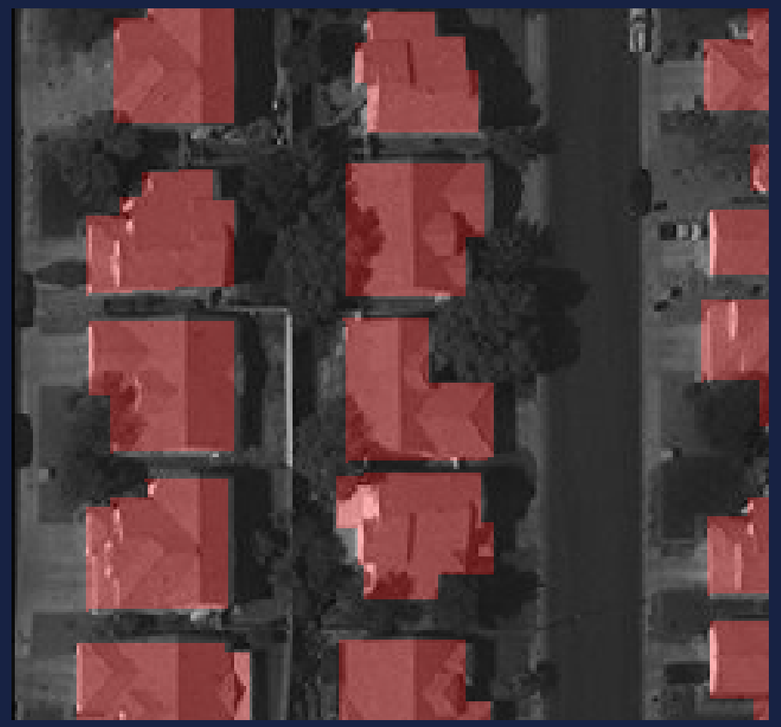
```
def generate_mask(encodings, labels):
    mask = np.zeros(array= , dtype= )

    for encoding, label in zip(encodings, labels):
        index = label - 1
        mask [:,:,index] = mask_rebuild(encoding).T

  return mask
```

GENERATION OF THE MASK

The returned array when passed through the *mask_rebuild* function along with a tuple specifying the dimension, returns the mask of the specified size as a numpy array.

```
def mask_rebuild(encoding, shape):
    run_arr = np.asarray([int(run) for run in encoding.split(' ')])
    run_arr[1::2] += run_arr[0::2]
    run_arr -= 1
    starting, ending = run_arr[0::2], run_arr[1::2]

    height, width = shape
    mask = np.zeros(height*width, dtype = uint8)
    for start, end in zip(starting, ending):
        mask[start:end] = 1
    return mask.reshape(shape)
```

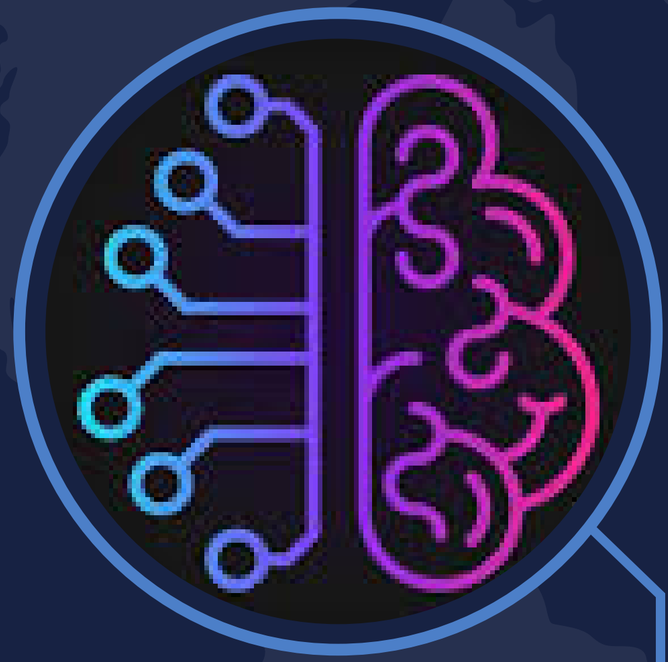
Areas selected for masking



The masked image

STEP:

CNN MODEL



ABOUT THE MODEL

CNN comprises of three kinds of layers:

1. The Convolutional layer is the core building block of a CNN, and it is where the majority of computation occurs. It requires a few components, which are input data, a filter, and a feature map.

2. Pooling layer aims to decrease number of parameters in input. Again, we use a Kernel to swipe over input but this time it does not learn any parameter instead it applies some aggregation function over the receptive field.

3. Fully Connected layer connects each node in the output layer directly to a node in the previous layer (like layers in ANN). While convolutional and pooling layers tend to use ReLu functions, FC layers usually leverage a softmax activation function to classify inputs appropriately, producing a probability from 0 to 1

OUR MODEL & CODE

USING KERAS FUNCTIONAL API

Making an input node of image size + (3,) i.e if image size is 32 X 32 then our input node will be of size 32 X 32 X 3 (for rgb channels).

```
ip_s = keras.Input(shape=img_size + (3,))
```

Spatial convolution over samples

```
x = layer.Conv2D(filters=32, kernel_size=(3, 3), strides=(2, 2), padding="same")(ip_s)
```

32 filters , each of size 3 x 3 is used , while keeping padding as "same" . Since , our strides is (2,2) , it will decrease our input size.

Now we will normalize the input recieved from previous layer using batch normalization .
This will help aginst overfitting of data as well as it will faster the training process

If a node will remain active in model is depends on its summed weighted input . The rectified linear activation function or ReLU for short is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero.

```
x = layer.BatchNormalization()(x)
# Rectified linear unit is used as activation function
x = layer.Activation("relu")(x)
```

Instead of having normal convolution layers in our model, we are including layers having Depthwise Seperable Convolution and then pointwise convolution. This will helps us in reducing computational complexity as well as faster training. layer.SeparableConv2D is doing Depthwise Seperable Convolution

```
prev_block_activation = x

for filters in arr:
    # arr contains different values of filters
    x = layer.Activation("relu")(x)
    # spatial convolution followed by pointwise convolution
    x = layer.SeparableConv2D(filters, kernel_size=(3, 3), padding="same")(x)
    x = layer.BatchNormalization()(x)

    x = layer.Activation("relu")(x)
    x = layer.SeparableConv2D(filters, kernel_size=(3, 3), padding="same")(x)
    x = layer.BatchNormalization()(x)

    # Using max pooling
    # on 2D spatial data
    x = layer.MaxPooling2D(pool_size=, strides=(2, 2), padding="same")(x)
    res = layer.Conv2D(filters, kernel_size=(1, 1), strides=(2, 2), padding="same")
(prev_block_activation)
    x = layer.add([x, res])
    # Update value
    prev_block_activation = x
```

After doing downsampling in our model, we will do Upsampling. We are using Nearest Neighbour and Transposed Convolution for Upsampling.

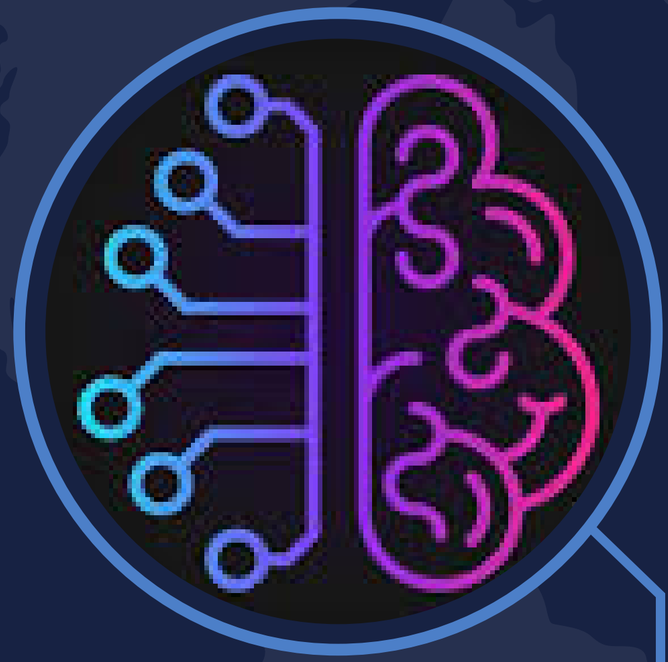
```
# Upsampling
for filters in arr:
    # arr contains different values for filters
    x = layer.Activation("relu")(x)
    x = layer.Conv2DTranspose(filters, kernel_size=(3, 3), padding="same")(x)
    x = layer.BatchNormalization()(x)

    x = layer.Activation("relu")(x)
    x = layer.Conv2DTranspose(filters, kernel_size=(3, 3), padding="same")(x)
    x = layer.BatchNormalization()(x)

    x = layer.UpSampling2D(2)(x)

    res = layer.UpSampling2D(2)(prev_block_activation)
    res = layer.Conv2D(filters, kernel_size=(1, 1), padding="same")(res)
    x = layer.add([x, res])
    prev_block_activation = x
```

Finally creating a fully connected layer using softmax activaion

```
op_s = layer.Conv2D(N, (3, 3), activation="softmax", padding="same")(x)
```

STEP:

CREATING A U-NET MODEL

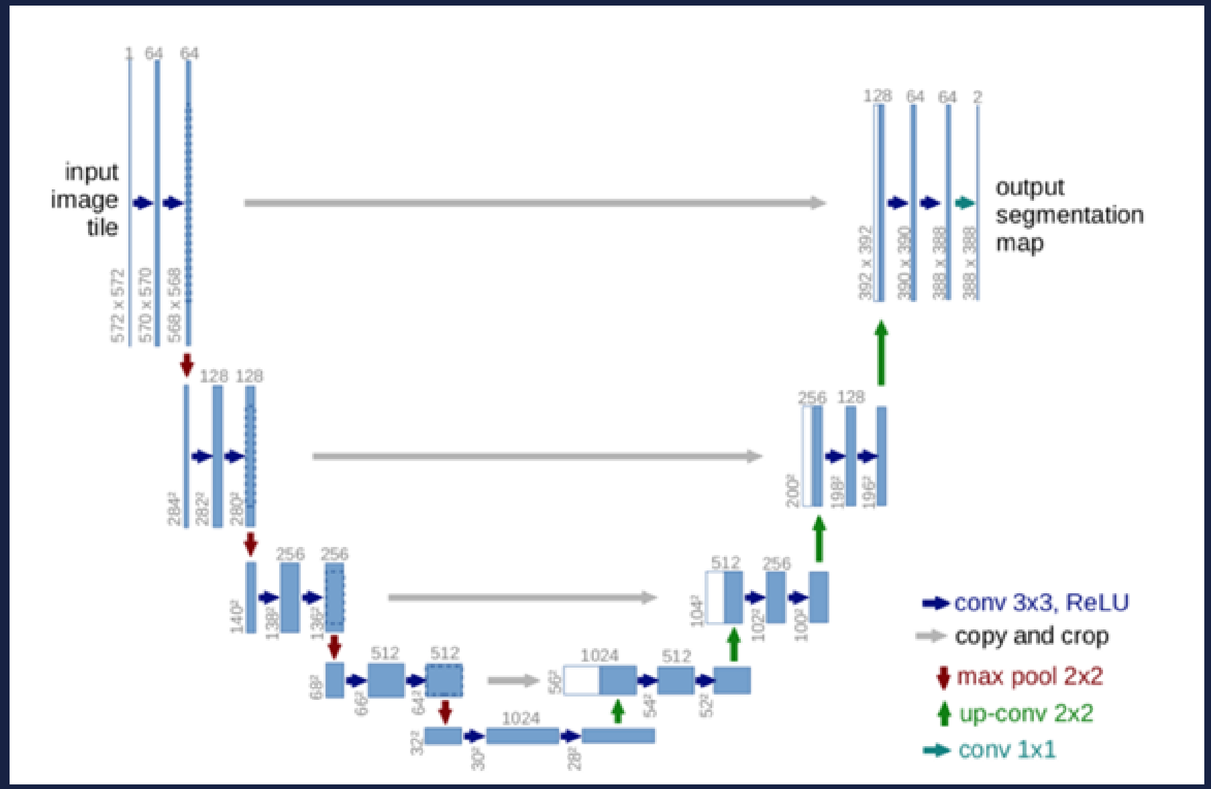


ABOUT THE MODEL

The u-net model is a convolutional neural network that takes an input image and outputs an image of the same size. Imagine the image to be a 3d matrix specifying pixels values, having dimensions (height * width * depth).

The architecture of this network is U-shaped. It is symmetric and consists of two major parts — the left part is called contracting path, which is constituted by the general convolutional process; the right part is expansive path, which is constituted by upsampling blocks.

Then we have several upsampling operations which decrease the depth and increase width and height of the image. After the final upsampling, the image output dimensions are the same as the input dimensions.



OUR MODEL & CODE

WE HAVE USED TENSORFLOW.KERAS.LAYERS

We have 5 downsampling blocks. The first block takes the input image. In each downsampling block, we have two 3x3 convolutions, each convolution followed by batch normalization and activated with ‘elu’. Finally we have a max pooling operation which reduces the height and width of the image for next block.

The third downsampling block is shown below:

```
conv3 = layer.Convolution2D(128, 3, 3, border_mode='same', init='he_uniform')(pool2)
conv3 = layer.normalization.BatchNormalization(mode=0, axis=1)(conv3)
conv3 = layer.advanced_activations.ELU()(conv3)
conv3 = layer.Convolution2D(128, 3, 3, border_mode='same', init='he_uniform')(conv3)
conv3 = layer.normalization.BatchNormalization(mode=0, axis=1)(conv3)
conv3 = layer.advanced_activations.ELU()(conv3)
pool3 = layer.MaxPooling2D(pool_size=(2, 2))(conv3)
```

Next we have 4 upsampling blocks. In each upsampling block, we combine the final layer of previous block with a final layer (the last layer before max pooling) of a downsampling block (which is at same level), and this produces a layer with increased height and width. Now we again have two 3x3 convolutions, each convolution followed by batch normalization and activated with ‘elu’, on this upscaled layer. For the last upsampling block, the final layer is subjected to a 1x1 convolution operation with ‘sigmoid’ activation. The ‘sigmoid’ activation produces all values in the range of [0,1].

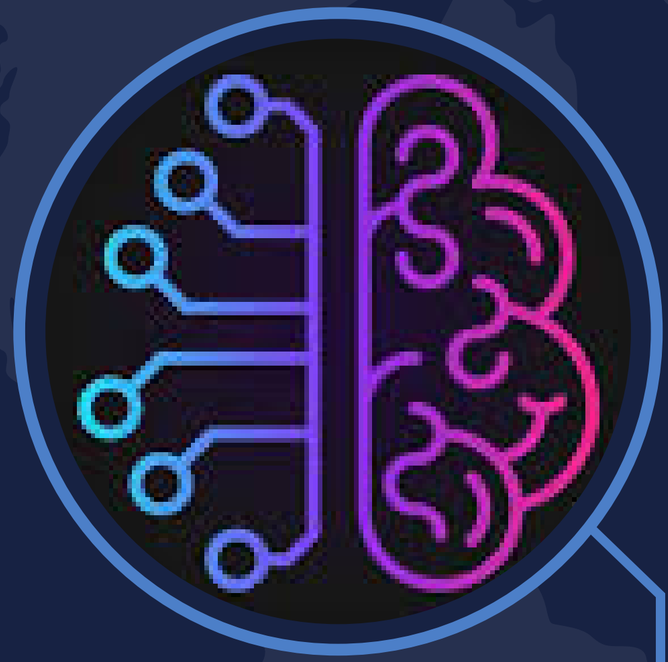
The second upsampling block is shown below:

```
up7 = merge([UpSampling2D(size=(2, 2))(conv6), conv3], mode='concat', concat_axis=1)
conv7 = layer.Convolution2D(128, 3, 3, border_mode='same', init='he_uniform')(up7)
conv7 = layer.normalization.BatchNormalization(mode=0, axis=1)(conv7)
conv7 = layer.advanced_activations.ELU()(conv7)
conv7 = layer.Convolution2D(128, 3, 3, border_mode='same', init='he_uniform')(conv7)
conv7 = layer.normalization.BatchNormalization(mode=0, axis=1)(conv7)
conv7 = layer.advanced_activations.ELU()(conv7)
```

Finally we create the model using the input layer (input image), and the final layer of the final upscaling block. The model can then be compiled and trained on our dataset.

STEP:

IMAGE SEGMENTATION



ABOUT THE MODEL

Masked images are taken as input of the model , output of our already implemented model (U-net).

The model can be tested on labelled as well as unlabelled data.
Convolutional Nural network is used for output model prediction.
Activation with Exponential Linear Unit is used in CNN model.

Keras backend is used to determine the true positive rate (also called sensitivity).

Dice coefficient is used to evaluate the prediction.

SEGMENTING IMAGES AND DATASET SPLITTING :

Before implementation of the neural network :

At the beginning, few random images are taken on which we will test our model.
A dataframe of images is created and annotation is done .

We segment the images, and then split the dataset.

we generate batch image taking Batch size =8

THE NEURAL NETWORK

Activation method :

We use exponential linear unit (ELU) for Activation . This speeds up learning in deep neural networks and leads to higher classification accuracies. ELUs alleviate the vanishing gradient problem via the identity for positive values and have negative values which allows them to push mean unit activations closer to zero like batch normalization but with lower computational complexity, that speed up learning because of a reduced bias shift effect .

Gaussian noise :

We added additive zero-centered Gaussian noise to images .
This is useful to mitigate overfitting .
Is statistical noise having a PDF equal to that of the Gaussian distribution.

THE NEURAL NETWORK

Batch Normalisation :

We used Batch normalization to apply a transformation that maintains the mean output close to 0 and the output standard deviation close to 1. During inference, the layer normalizes its output using a moving average of the mean and standard deviation of the batches it has seen during training .

Returns :
(batch - self.moving_mean) / (self.moving_var + epsilon) * gamma + beta
Where ,
moving_mean = moving_mean * momentum + mean(batch) * (1 - momentum)
moving_var = moving_var * momentum + var(batch) * (1 - momentum)

[the layer will only normalize its inputs during inference after having been trained on data that has similar statistics as the inference data.]

OUR MODEL & CODE

Evaluation of the model :

The prediction we are getting is evaluated by Dice Coefficient for comparing algorithm output against reference masks

Dice Coefficient = (2 * the Area of Overlap) /(total number of pixels in both images)

Given two sets, X and Y, it is defined as :

2* |X intersection Y|

|X| + |Y|

The function ranges between zero and one.

This can be considered a semimetric version of the Jaccard index.

```
def dice_coef(y_true, y_pred, smooth=1):
        intersection = K.sum(y_true * y_pred, axis=[1, 2, 3])
        union = K.sum(y_true, axis=[1, 2, 3]) + K.sum(y_pred, axis=[1, 2, 3])
        return K.mean((2. * intersection + smooth) / (union + smooth), axis=0)
```

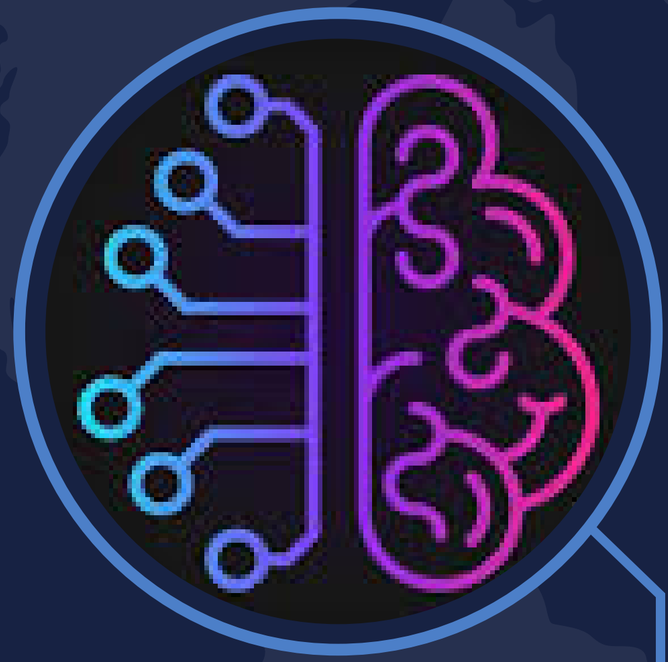
OPTIMISER AND LOSS FUNCTION

The model is configured for training using the Adam optimiser. Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments.

Loss function for compilation of the model is calculated with help of binary crossentropy.

```
def dice_p_bce(in_gt, in_pred):
        return 0.05 * binary_crossentropy(in_gt, in_pred) - dice_coef(in_gt, in_pred)

seg_model.compile(optimizer=Adam(1e-4, decay=1e-6), loss=dice_p_bce, metrics=[dice_coef,
'binary_accuracy', true_positive_rate])
```

OUR FINAL MODEL

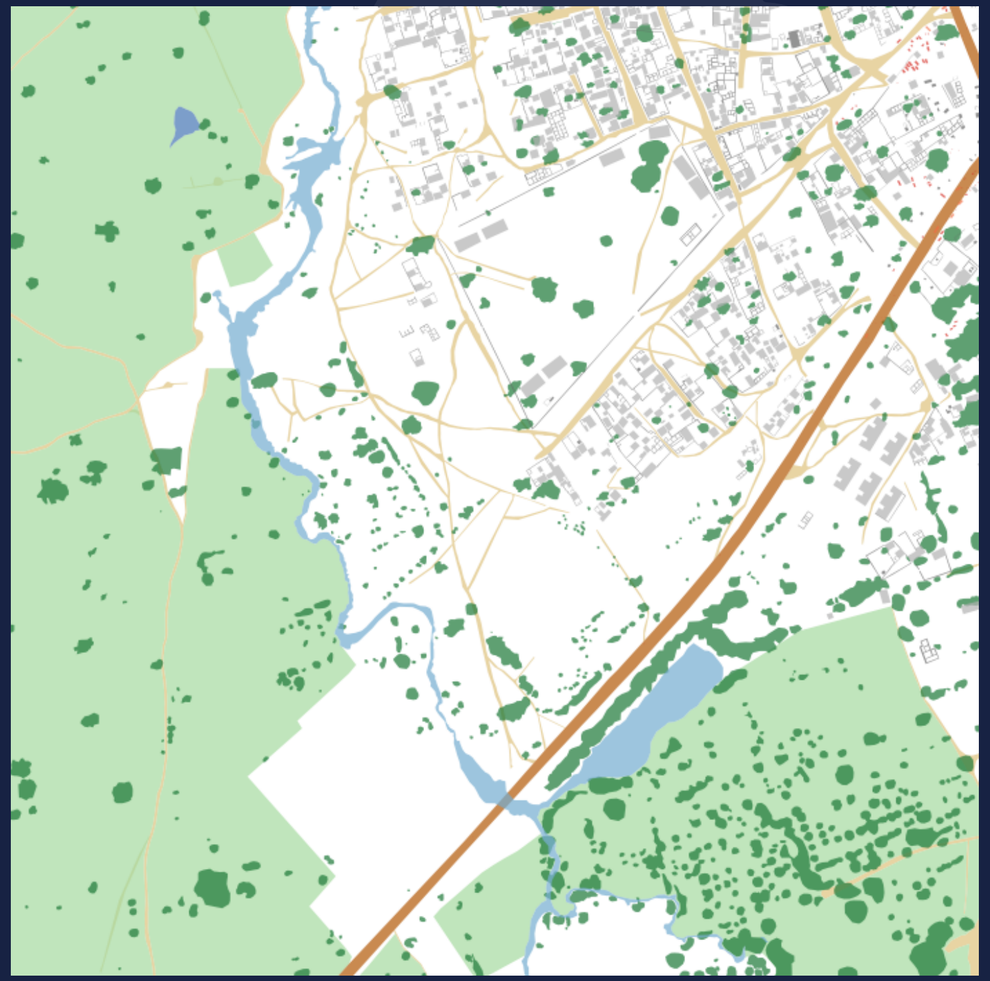


The final model was built using the masking and convolutional neural network. The parameters were changed many times which is why there are many versions available for the final model. In few cases we applied some already available models or their neural network architecture.

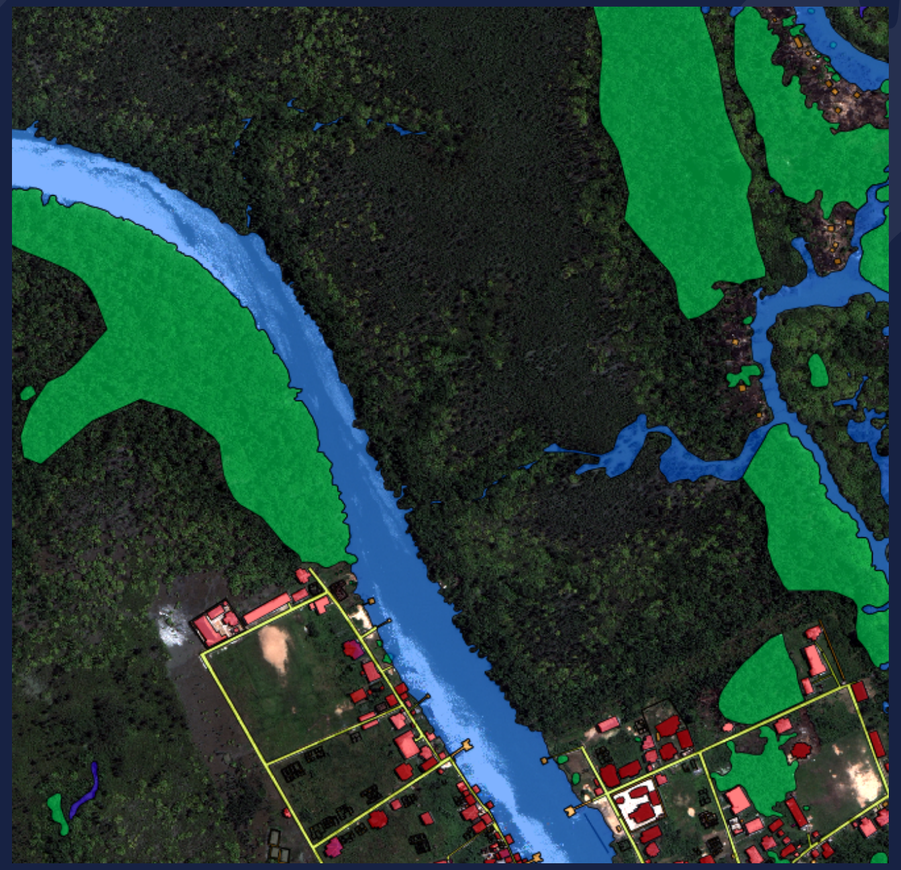
The code for building segmentation model can be used as a template for the final model.

The CNN used here is almost the same as in the building segmentation model.

FINAL IMAGES



FINAL IMAGES



Dataset and resources used

- **DSTL SATELLITE IMAGE DATASET**
  https://www.crowdai.org/challenges/mapping-challenge

- **KERAS FUNCTIONAL API**
  https://keras.io/guides/functional_api/

- **BATCH NORMALIZATION BLOG**
  https://www.analyticsvidhya.com/blog/2021/03/introduction-to-batch-normalization/

- **ACTIVATION FUNCTION BLOG**
  https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/

- **TRANSPOSED CONVOLUTION BLOG**
  https://towardsdatascience.com/transposed-convolution-demystified-84ca81b4baba

- **SEPARABLE CONVOLUTIONS BLOG**
  https://towardsdatascience.com/a-basic-introduction-to-separable-convolutions-b99ec3102728

Thank you...