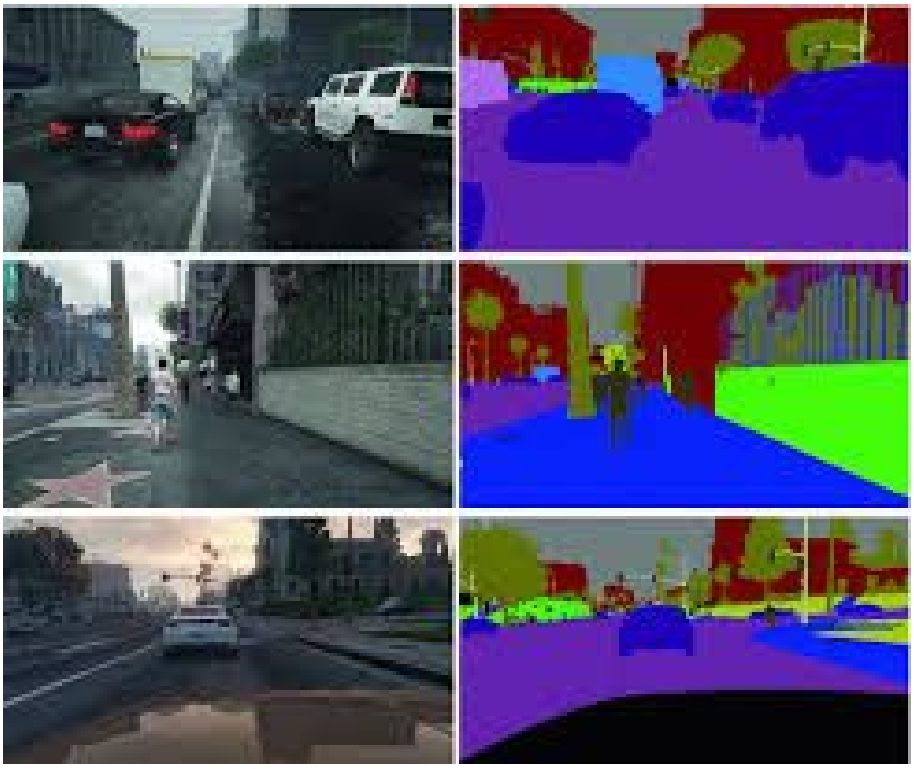
Image Segmentation



Introduction

Image segmentation is the process of assigning labels to subsets of the image in such a way that subsets with the same label typically have some characteristics in common while adjacent subsets will be significantly different in regards to the same characteristics.

Use cases

Image processing has a wide variety of uses. The most notable being: medical imaging like tumor detection and other pathologies, object detection, recognition and traffic control systems.

Techniques for segmentation (basic ones)

There are various techniques to process and segment different areas on the image, some of the most well studied ones are:

Thresholding

This technique converts grayscale images into a binary image map, by setting some threshold (T) if the pixel value is higher than this threshold the image pixel is converted to white else it is converted to black. One can also set multiple thresholds for various segments within the image. Some limitations to this approach. When we don't have significant grayscale difference, or there is an overlap of the grayscale pixel values, it becomes very difficult to get accurate segments.

Balanced image thresholding

This technique is similar to the one mentioned above, but in this case the threshold is determined automatically by balancing the histogram of pixel values.

Clustering

This technique makes use of any clustering technique (the most notable being KMeans). This algorithm will cluster pixel values together based on pixel values with similar characteristics/values into k clusters - this segmenting the image into different parts.

Kernels

These form the basis of convolutional networks as they help to detect edges within an image. The most notable being Sobel edge detection for both horizontal and vertical edges. Typically these kernels are generated with a random set of values and as part of back propagation these values in the kernels are updated during the learning phase to detect edges for the particular data set.

Advanced deep learning methods

Research has improved over the years and various alternatives has been introduced. Some of the most notable techniques will be briefly mentioned here.

Fully Convolutional Network (FCN)

This method applies a series of convulations to downsample the image to a smaller size, this is called the encoding phase, thereafter a transpose set of convolutions are applied to upsample the output this is called the decoding phase. There were several drawbacks with this approach one of these had to do with the poor resolution of the image at the boundaries due to the loss of information during the encoding phase. Subsequently other techniques go introduced.

U-Net

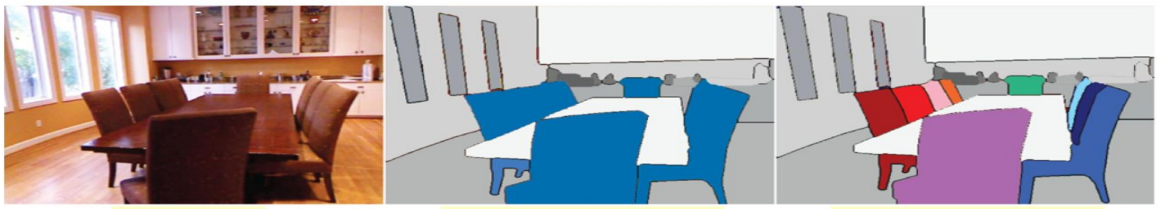
This model is an improvement on the FCN model and was originally developed and used for biomedical image segmentation. This model architecture has skip connections that allows gradients to flow better and provides information from multiple scales of the image size.

Mask-RCNN

Current state of the art algorithm for instance segmentation it is a two-stage approach with multiple sub-networks working together: RPN (Region Proposal Network), FPN (Feature Pyramid Network) and FCN (Fully Convolutional Network).

Different types of segmentation

There are two types of segmentation problems: the one is semantic segmentation and the other is instance based segmentation. Semantic segmentation treats multiple objects of the same class as a single entity (not to be confused with classification that labels the whole image as belonging to a class). On the other hand, instance segmentation treats multiple objects of the same class as distinct individual objects. The latter problem is harder to solve. The image below will make the distinction clear.



Semantic Segmentation vs Instance Segmentation

Segmentation loss function

There are a couple of loss functions available to use during the optimisation phase of the segmentation problem two of the most widely used loss functions are the pixel wise cross entropy function as segmentation can be formulated as a multiclass problem. Alternatively the most used loss due to its ability to be applied in the event of even huge class imbalances is called the dice loss.

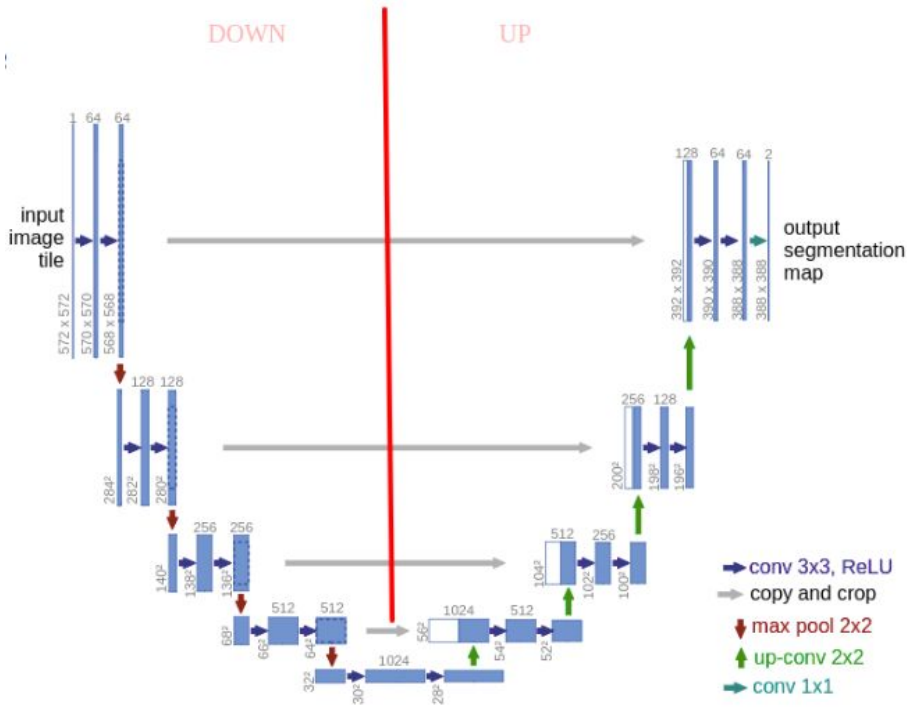
Dice Loss is used to calculate the overlap between the predicted class and the ground truth class. The Dice Coefficient (D) is represented as follows:

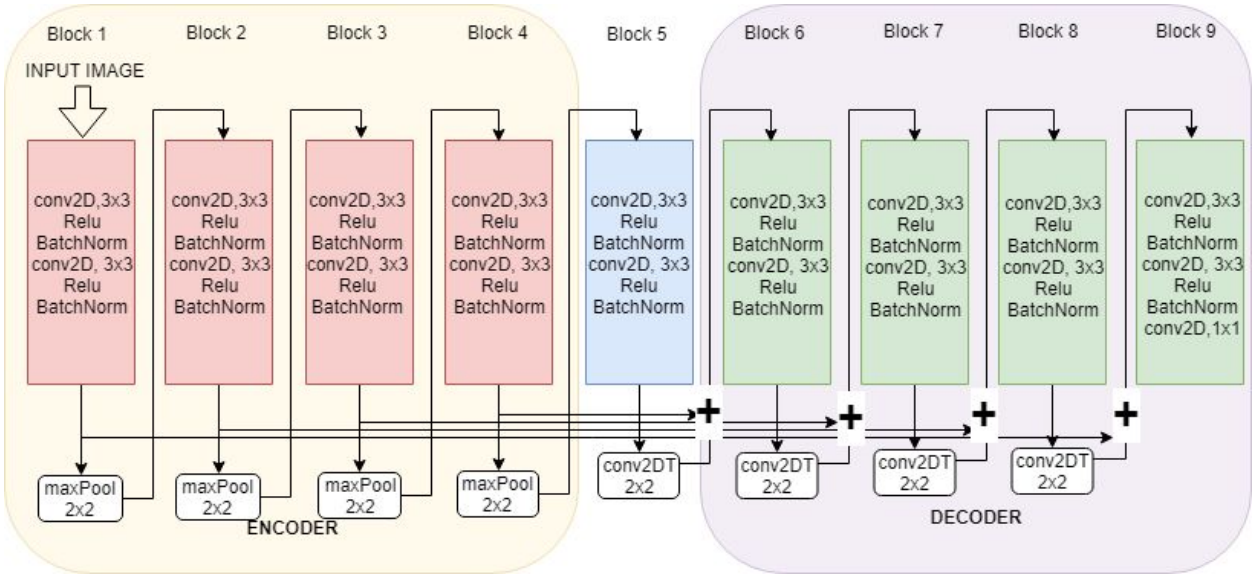
$$\frac{2|X \cap Y|}{|X| + |Y|}$$

The objective is to maximize the overlap between the predicted and ground truth class (i.e. to maximize the Dice Coefficient). Hence, we generally minimize (1-D) instead to obtain the same objective, as most ML libraries provide options for minimization only.

Dive into U-Net

The U-Net architecture as the name suggests has two sections to it - the first half of the U is the encoder part where you apply convolution blocks followed by a maxpool downsampling to encode the input image into feature representations at multiple different levels. The second part of the network consists of upsample and concatenation followed by regular convolution operations. We are expanding the feature dimensions to meet the same size with the corresponding concatenation blocks from the left. The main contribution of U-Net in this sense compared to other fully convolutional segmentation networks is that while upsampling and going deeper in the network we are concatenating the higher resolution features from down part with the upsampled features in order to better localize and learn representations with following convolutions.





The contraction section is made of many contraction blocks (block 1 to block 4). Each block takes an input and applies two 3X3 convolution layers followed by a 2X2 max pooling. The number of kernels or feature maps after each block doubles so that architecture can learn the

complex structures effectively. The bottom most layer (block 5) mediates between the contraction layer (encoder) and the expansion layer (decoder). It uses two 3X3 CNN layers followed by 2X2 up convolution layer (transpose convolution).

The heart of this architecture lies in the expansion section. Similar to contraction layer, it also consists of several expansion blocks (block 6 to block 9). Each block passes the input to two 3X3 CNN layers followed by a 2X2 upsampling layer (transpose convolution). Every time the input also get appended by feature maps of the corresponding contraction layer. This action would ensure that the features that are learned while contracting the image will be used to reconstruct it. The number of expansion blocks is the same as the number of contraction blocks. After that, the resultant mapping passes through another 3X3 CNN layer with the number of feature maps equal to the number of segments desired (this is the final output at block 9).

Implementation

In this section of the blog I will briefly explain the most important parts of a project where I used U-net architecture for image segmentation. This blog is not going to dive deep into the exploratory data analysis as this is covered in detail within the compiled report.

Project Summary

Severstal is leading the charge in efficient steel mining and production. They believe the future of metallurgy requires development across the economic, ecological, and social aspects of the industry—and they take corporate responsibility seriously. The company recently created the country's largest industrial data lake, with petabytes of data that were previously discarded. Severstal is now looking to machine learning to improve automation, increase efficiency, and maintain high quality in their production.

The production process of flat sheet steel is especially delicate. From heating and rolling, drying and cutting, several machines touch flat steel by the time it's ready to ship. Today, Severstal uses images from high frequency cameras to power a defect detection algorithm. The work presented in this paper is to help improve the algorithm by localizing and classifying surface defects on a steel sheet.

Creating masks

Instead of providing an exhaustive list of pixel positions of where the defects occur within the image a run length encoding (rle) is provided for each image. This is basically a string depicting the pixel start position and the length of positions that the defect runs across. As a first major component mask-to-image and image-to-mask conversion functions had to be developed.

```
def mask2rle(mask):
    '''
    mask: numpy array, 1 - mask, 0 - background
    Returns run length as string formated
    '''
    pixels = mask.T.flatten()
    pixels = np.concatenate([[0], pixels, [0]])
    runs = np.where(pixels[1:] != pixels[:-1])[0] + 1
    runs[1::2] -= runs[::2]
    return ' '.join(str(x) for x in runs)

def rle2mask(mask_rle, shape=(1600,256)):
    '''
    mask_rle: run-length as string formated (start length)
    shape: (width,height) of array to return
    Returns numpy array, 1 - mask, 0 - background

    '''
    if len(mask_rle) == 0:
        return np.zeros(shape[0]*shape[1], dtype=np.uint8).reshape(shape).T
    else:
        s = mask_rle.split()
        starts, lengths = [np.asarray(x, dtype=int) for x in (s[0:][::2], s[1:][::2])]
        starts -= 1
        ends = starts + lengths
        mask = np.zeros(shape[0]*shape[1], dtype=np.uint8)
        for lo, hi in zip(starts, ends):
            mask[lo:hi] = 1
        mask = mask.reshape(shape).T
        return mask

try:
    assert '1 3 10 5' == mask2rle(rle2mask('1 3 10 5'))
    assert '1 1' == mask2rle(rle2mask('1 1'))
    print('Function mask is good')
except AssertionError as e:
    print('Error in function mask')
```

Defining the custom dice loss function

```
def dice_coef(y_true, y_pred, smooth=1):
    y_true_f = np.ravel(y_true)
    y_pred_f = np.ravel(y_pred)
    intersection = np.sum(y_true_f * y_pred_f)
```

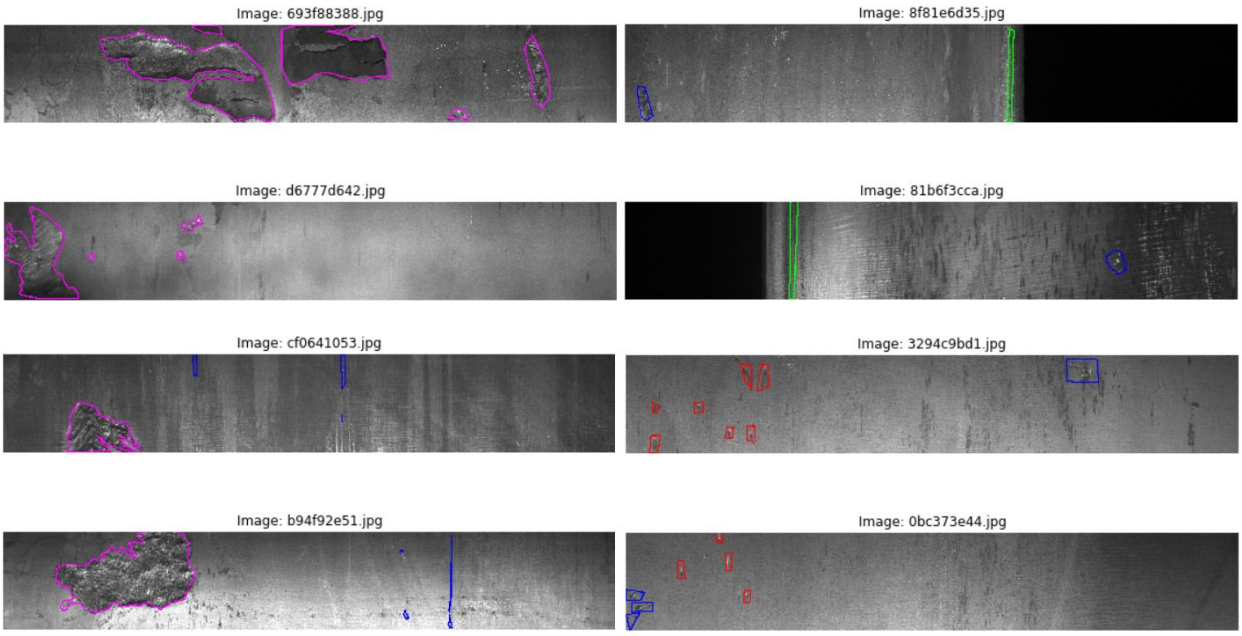
```
    return (2. * intersection + smooth) / (np.sum(y_true_f) + np.sum(y_pred_f) + smooth)

def dice_loss(y_true, y_pred, smooth=1):
    y_true_f = np.ravel(y_true)
    y_pred_f = np.ravel(y_pred)
    intersection = y_true_f * y_pred_f
    score = (2. * np.sum(intersection) + smooth) / (np.sum(y_true_f) + np.sum(y_pred_f) + smooth)
    return 1. - score
```

Mask overlays

```
def mask_to_contours(image, mask, color):
    """ Converts a mask to contours using OpenCV and draws it on the image
    """
    contours, hierarchy = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    cv2.drawContours(image, contours, -1, color, 2)
    return image

def apply_masks(file_name, encodings, labels):
    if not isinstance(encodings,list):
        encodings = list()
    if not isinstance(labels,list):
        labels = list()
    # reading in the image
    image = cv2.imread(os.path.join('./data/train_imgs',file_name))
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    masks = [rle2mask(rle) for rle in encodings]

    for mask,label in zip(masks,labels):
        image = mask_to_contours(image=image, mask=mask, color=color_scheme.get(label,(0,0,0)))
    return image
```

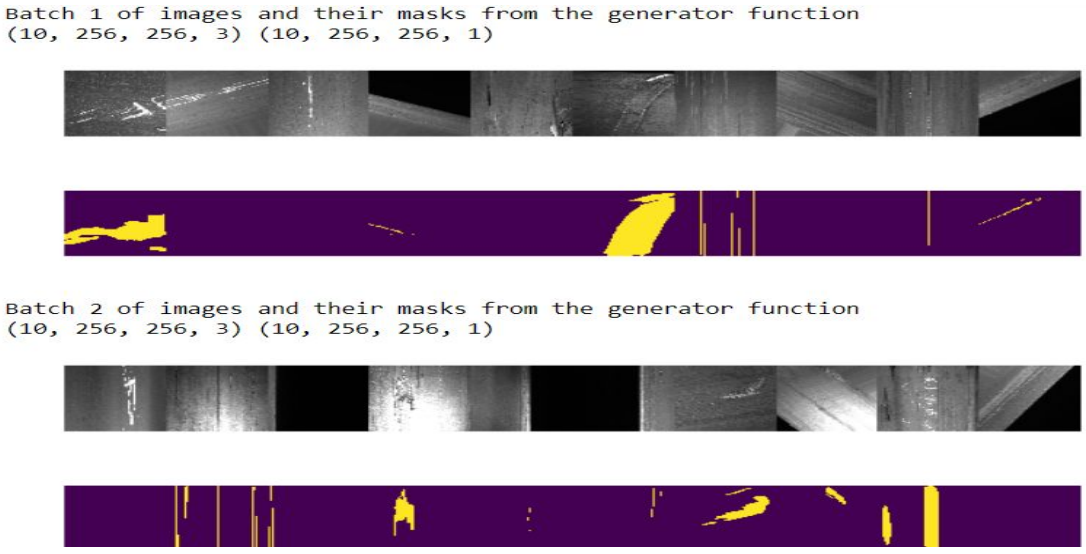
Visuals showing the defects

The images below show the steel with the type of defect that occur on it. Some of the image had no defects while others had a combinations of defects, making this a multiclass classification problem using segmentation to divide the image up into various segments or classes of defects.



Datagenerator

In order to achieve memory efficiency with such deep neural networks one has to make use of a data generator to yield batches and pass these batches through the network instead of training on the whole image set at once. Another key aspect is that the generator also extended to make use of data augmentation in order to perform random transformations as a means to help regularize and stabilize the model to prevent over fitting. Herewith the results of the images along with their masks after generating a batch using the generator. The batch size was set to 10 and these images all concatenated together (please not the projections and rotation of the images).



U-Net model architecture

The model as been implemented using the Keras API. Please see the image above in order to follow along with the code below. The notes on the code references back to the image.

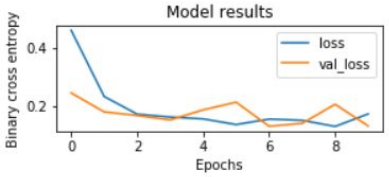
```
########################################-IMAGE INPUT-########################################
inputs = Input(shape=(256,256,3))


#####################################-ENCODER BLOCK-START-#####################################
# Block 1:
c1 = Conv2D(16, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (inputs)
c1 = Dropout(0.1) (c1)
c1 = Conv2D(16, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (c1)
p1 = MaxPooling2D((2, 2)) (c1)

# Block 2:
c2 = Conv2D(32, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (p1)
c2 = Dropout(0.1) (c2)
c2 = Conv2D(32, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (c2)
p2 = MaxPooling2D((2, 2)) (c2)

# BLock 3:
c3 = Conv2D(64, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (p2)
c3 = Dropout(0.2) (c3)
c3 = Conv2D(64, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (c3)
p3 = MaxPooling2D((2, 2)) (c3)

# Block 4:
c4 = Conv2D(128, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (p3)
c4 = Dropout(0.2) (c4)
c4 = Conv2D(128, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (c4)
p4 = MaxPooling2D(pool_size=(2, 2)) (c4)

#####################################-ENCODER BLOCK END-#####################################

# Block 5:
c5 = Conv2D(256, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (p4)
c5 = Dropout(0.3) (c5)
c5 = Conv2D(256, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (c5)

# Concatenate block 4 output with the upsample:
u6 = Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='same') (c5)
u6 = concatenate([u6, c4])

#####################################-DECODER BLOCK START-#####################################
# Block 6:
c6 = Conv2D(128, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (u6)
c6 = Dropout(0.2) (c6)
c6 = Conv2D(128, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (c6)

# Concatenate block 3 output with the upsample:
u7 = Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same') (c6)
u7 = concatenate([u7, c3])
```

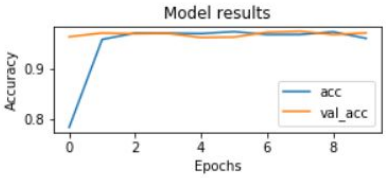
```
# Block 7:
c7 = Conv2D(64, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (u7)
c7 = Dropout(0.2) (c7)
c7 = Conv2D(64, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (c7)

# Concatenate block 2 output with the upsample:
u8 = Conv2DTranspose(32, (2, 2), strides=(2, 2), padding='same') (c7)
u8 = concatenate([u8, c2])

# Block 8:
c8 = Conv2D(32, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (u8)
c8 = Dropout(0.1) (c8)
c8 = Conv2D(32, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (c8)

# Concatenate block 1 output with the upsample:
u9 = Conv2DTranspose(16, (2, 2), strides=(2, 2), padding='same') (c8)
u9 = concatenate([u9, c1], axis=3)

# Block 9:
c9 = Conv2D(16, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (u9)
c9 = Dropout(0.1) (c9)
c9 = Conv2D(16, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (c9)

########################################-DECODER BLOCK END-########################################


##########################################-SEGMENTATION-##########################################
outputs = Conv2D(filters=1, kernel_size=(1,1), strides=(1,1), padding="same", activation='sigmoid') (c9)

model = Model(inputs=[inputs], outputs=[outputs])
```

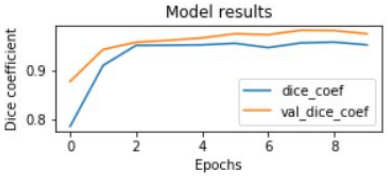
The model was compiled using Adam, loss binary cross entropy, metrics accuracy and dice loss.

Results

	val_loss	val_acc	val_dice_coef	loss	acc	dice_coef
0	0.245577	0.963855	0.877449	0.461042	0.782964	0.786119
1	0.180323	0.971314	0.942457	0.233302	0.958508	0.910008
2	0.167895	0.969900	0.957093	0.172597	0.971272	0.950833
3	0.152670	0.970522	0.960993	0.162738	0.971090	0.951103
4	0.187234	0.962297	0.965939	0.156291	0.970252	0.951740
5	0.213654	0.963289	0.974379	0.136794	0.974010	0.954925
6	0.131007	0.972849	0.972446	0.155614	0.968190	0.946230
7	0.140876	0.974925	0.981411	0.151404	0.968380	0.955725
8	0.206396	0.968225	0.980820	0.130366	0.973730	0.957365
9	0.132237	0.971559	0.974180	0.173319	0.960671	0.951788







Resources

- https://www.topbots.com/semantic-segmentation-guide/
- https://www.analyticsvidhya.com/blog/2019/04/introduction-image-segmentation-techniques-python/
- https://www.analyticsvidhya.com/blog/2019/07/computer-vision-implementing-mask-r-cnn-image-segmentation/?utm_source=blog&utm_medium=introduction-image-segmentation-techniques-python
- https://medium.com/@keremturgutlu/semantic-segmentation-u-net-part-1-d8d6f6005066
- https://towardsdatascience.com/review-deepmask-instance-segmentation-30327a072339
- https://github.com/ternaus/TernausNet/blob/master/unet_models.py
- https://github.com/matterport/Mask_RCNN
- https://medium.com/@mohamedalihabib7/brain-tumor-detection-using-convolutional-neural-networks-30ccef6612b0