Brain Abnormalities Image segmentation using Convolutional Neural networks

Project Report

Submitted towards the Course

AIDS-RB-20CS2221RB 2021-22 Even Semester

By

P. Jaswanth – 2000039034

BACHELOR OF TECHNOLOGY

Branch: Computer Science and Engineering

Department of Computer Science and Engineering
K L Deemed to be University,
Green Fields, Vaddeswaram,
Guntur District, A.P., 522 502.

Abstract

In this project, we try to address the problem of segmentation of tumors/abnormalities in Brain MRI images with the help of deep Learning i.e a Convolutional Neural Network in particular. The CNN model used here is known as a U-Net and the dataset we use to train and test our results is Brain MRI images together with manual FLAIR abnormality masks. The goal of the project is for us(group members) to gain experience on machine learning’s real life application and implementation.

Keywords: Machine Learning, Deep learning, Convolutional Neural Network, Brain-MRI

1) Introduction to problem statement:

Over the last few decades, the rapid development of noninvasive brain imaging technologies has opened new horizons in analyzing and studying the brain anatomy and function and also in the detection of abnormalities. Enormous progress in accessing brain injury and exploring brain anatomy has been made using magnetic resonance imaging (MRI). The advances in brain MR imaging have also provided a large amount of data with an increasingly high level of quality. The analysis of these large and complex MRI datasets has become a tedious and complex task for clinicians, who have to manually extract important information such as identifying brain abnormalities. This manual analysis is often time-consuming and prone to errors due to various inter- or intra operator variability studies. These difficulties in brain MRI data analysis required inventions in computerized methods to improve disease diagnosis and testing. Nowadays, computerized methods for MRI image segmentation, registration, and visualization have been extensively used to assist doctors in qualitative diagnosis.

Brain MRI segmentation is an essential task in many clinical applications because it influences the outcome of the entire analysis. This is because different processing steps rely on accurate segmentation of abnormal regions. For example, MRI segmentation is commonly used for measuring and visualizing different brain structures, for delineating lesions, for analyzing brain development, and for image-guided interventions and surgical planning. This diversity of image processing applications has led to development of various segmentation techniques of different accuracy and degree of complexity.

In this project we build a Deep learning model using U-net and keras and try to come up with an accurate segmentation tool which would be useful for something so important and complex. To introduce the reader to the complexity of the brain MRI segmentation problem and address its challenges, we first introduce describing an image segmentation problem and image features, the basic concepts of image segmentation with deep learning with the use of convolutional neural networks. Finally, after reviewing the segmentation method, we discuss the implementation of a neural network for brain MRI segmentation.

2) **Theoretical Background:**

2.a) Image Segmentation problem

One of the most important operations in Computer Vision is Segmentation. Image segmentation is the task of clustering parts of an image together that belong to the same object class. This process is also called pixel-level classification. In other words, it involves partitioning images (or video frames) into multiple segments or objects.

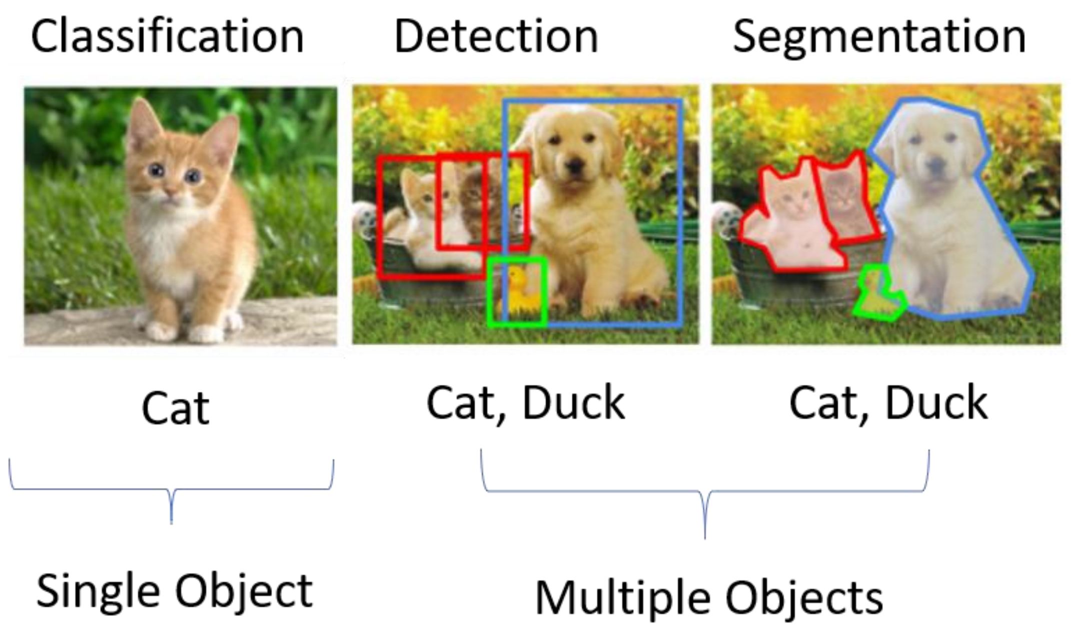
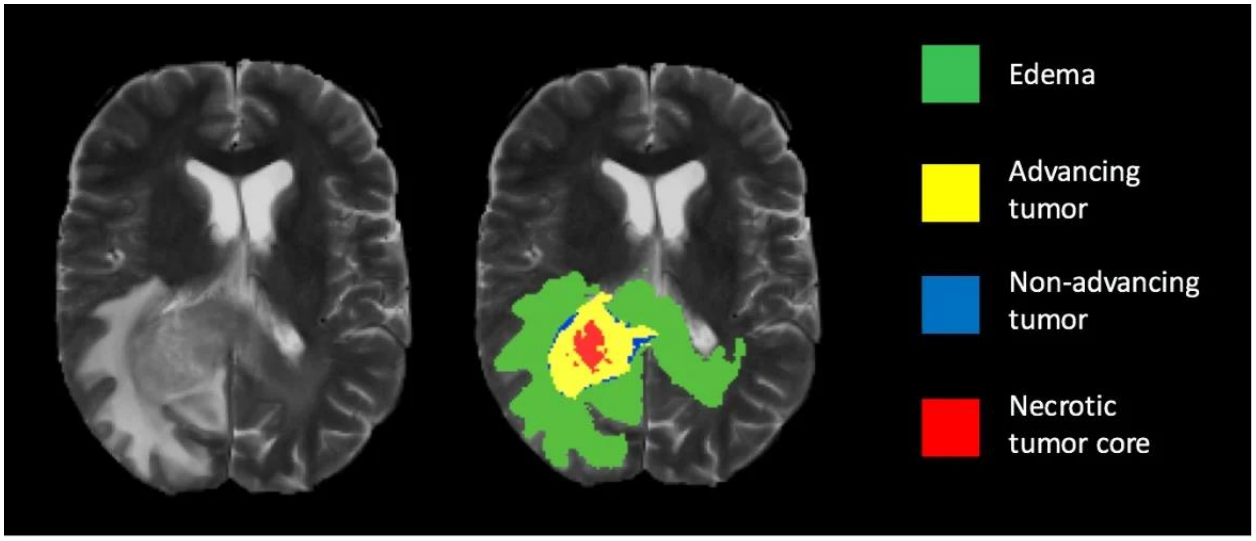


Image segmentation results in more granular information about the shape of an image and thus an extension of the concept of Object Detection.

We segment i.e. divide the images into regions of different colors which helps in distinguishing an object from the other at a finer level.

In the case of our project we use this image segmentation on medical images, Brain MRI images to be specific, an example for the Image segmentation is shown below.

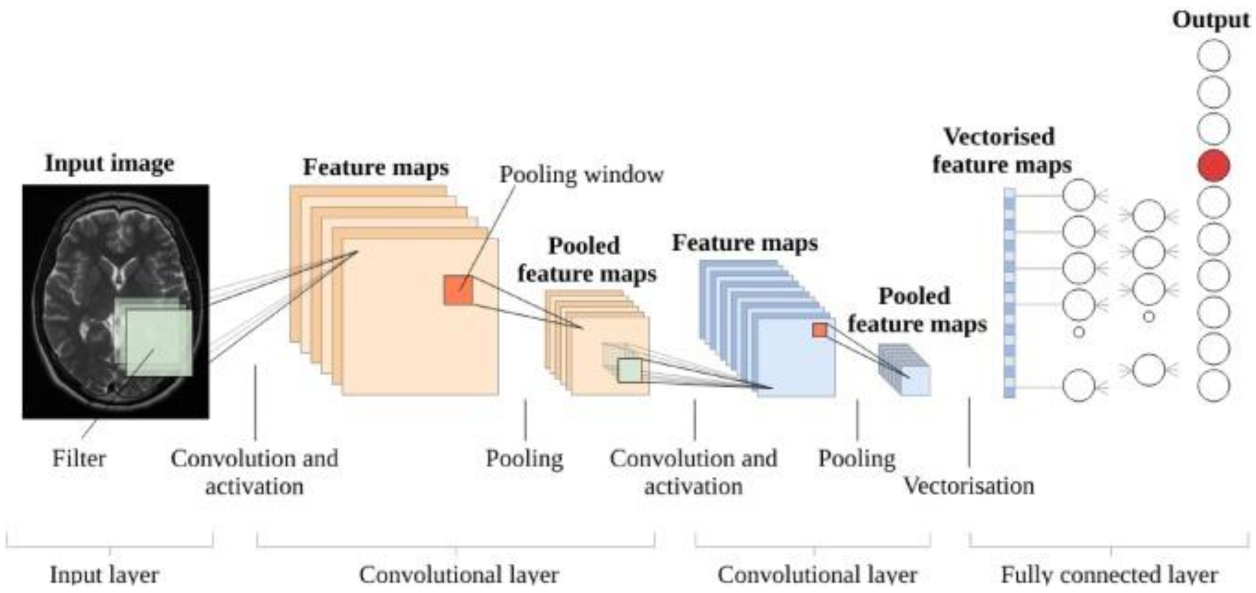


In this case, Image segmentation can be used to extract clinically relevant information from medical reports. For example, image segmentation can be used to segment tumors.

2.b) Convolutional neural networks:

A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics.

The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area.



Convolutional neural networks are distinguished from other neural networks by their superior performance with image, speech, or audio signal inputs. They have three main types of layers, which are:

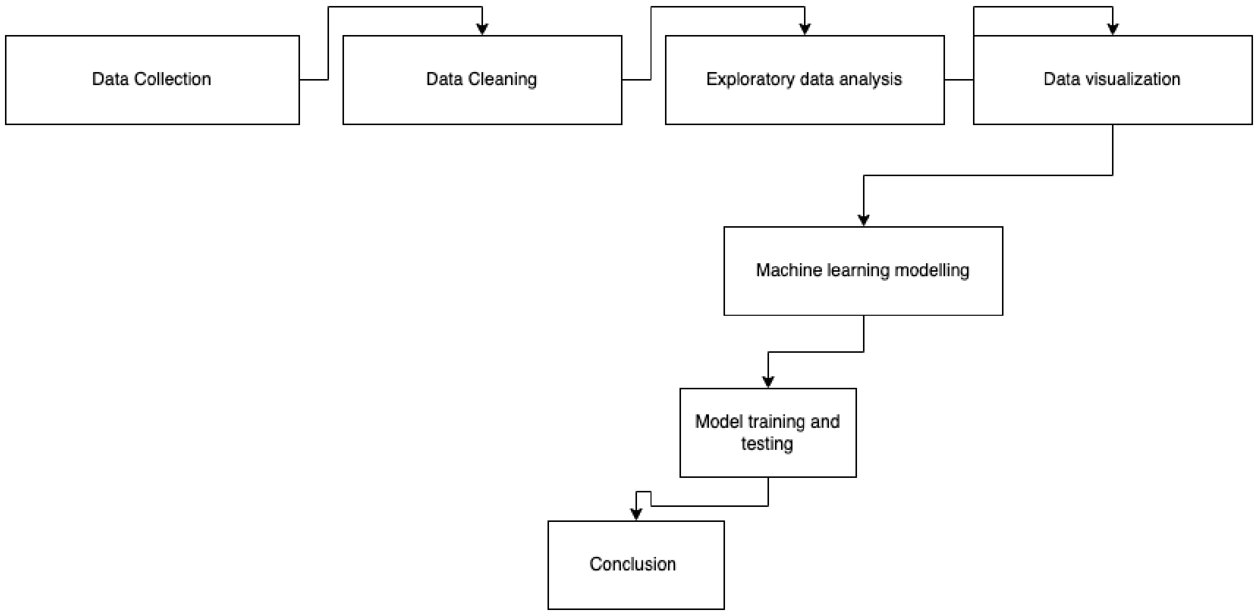
- Convolutional layer
- Pooling layer
- Fully-connected (FC) layer

3) Software Requirements:

Software: Anaconda, Jupyter Notebook/ Google collab
Programming Language: python 3.10
Dataset : LGG Segmentation Dataset
Space required: 2gb
Hardware requirements: MacOS, windows 7/8/10

4) **Project Flow Chart**

a project flow chart can be constructed as follows:


Data Collection
Data Cleaning
Exploratory data analysis
Data visualization
Machine learning modelling
Model training and testing
Conclusion

The data cleaning step in this project would be just to pick the important image-mask pairs from the data but rather not clean any of the images as this is a perfectly constructed database from the start. If any such case arises where we only find only the mask or only the image without the pair we discard such images

In the exploratory data analysis we try to understand the dataset by looking at which brain MRI images have MRI or Wether or not we have Brain MRI images which do not have tumors

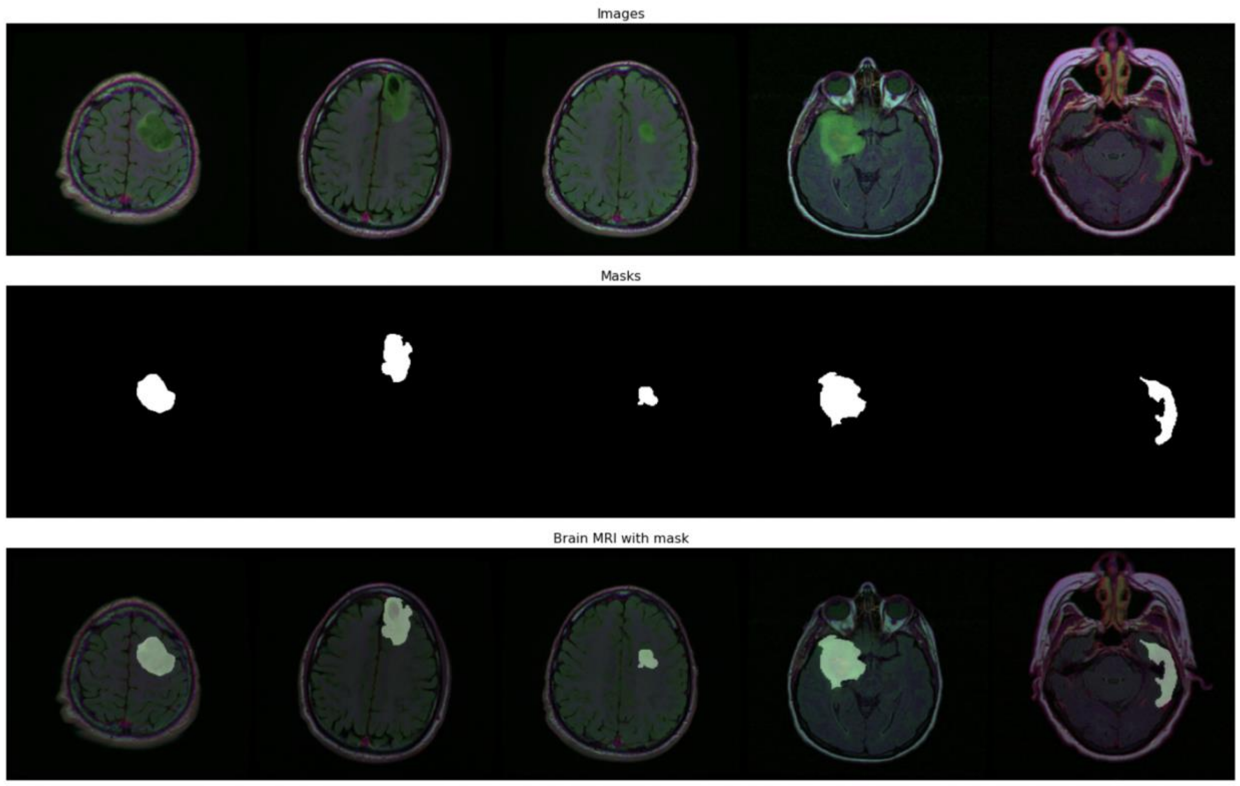
In data Visualization step we visualize the datasets images using the Python CV2 library. This library can also be used to further feed the machine learning model for training and testing

Here on onwards we focus on the Model implementation in this case how to implement the U-net architecture using Tensorflow and keras. And followed by this we check the accuracy and display the predictions

5) **Dataset:**

The dataset consists of Brain MRI images together with Manual Flair Abnormality segmentation masks. The images were obtained from The Cancer Imaging Archive (TCIA)[1] .They correspond to 110 patients included in The Cancer Genome Atlas (TCGA) lower-grade glioma collection with at least fluid-attenuated inversion recovery (FLAIR) sequence and genomic cluster data available. Tumor genomic clusters and patient data is provided in data.csv file.

The image-mask pair are stored in tif format. A few examples of the image-mask combined and plotted using CV2 library is as shown



In total there are **3929** pairs of image-mask’s in the dataset. These high quality images are used for further exploratory data analysis and followed by the machine learning model implementation.

6) Data Exploration:

As it after displaying the data we can see that for some of Brain Images we have empty mask, in order to better understand the data it is better to plot the fraction of datasets having no tumors.

The following code achieves the task by trying to map a dataframe based on the value of the maximum pixel value present in the data. If the mask pixels have a maximum value of 0 then the corresponding brain MRI does not have a tumor. If the maximum value of the mask pixels is 1 or 2 then the corresponding Brain MRI Image has a tumor.

```python
def diagnosis(mask_path):
    value = np.max(cv2.imread(mask_path))
    return '1' if value > 0 else '0'
df = pd.DataFrame({"image_path": train_files,
                   "mask_path": mask_files,
                  "diagnosis":[diagnosis(x) for x in mask_files]})
df.head()
```

Here the diagnosis refers to having tumor(1) and not having tumor (0). The respective pandas dataframe created would be of the following form

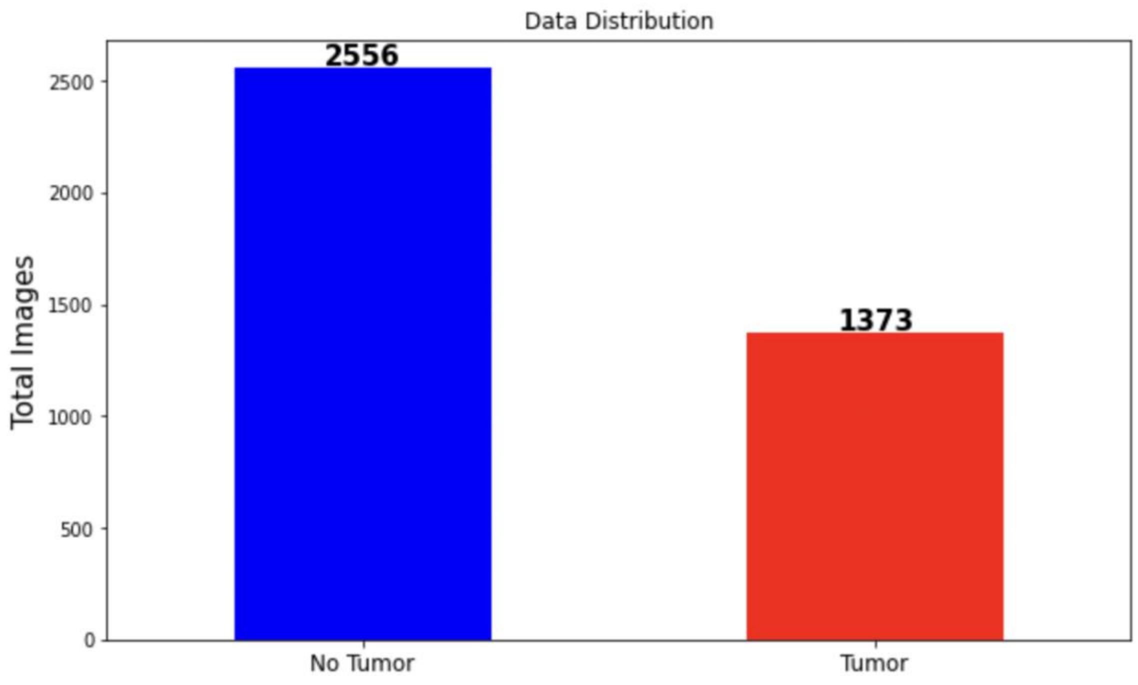
	image_path	mask_path	diagnosis
0	./kaggle_3m\TCGA_CS_4941_19960909\TCGA_CS_4941...	./kaggle_3m\TCGA_CS_4941_19960909\TCGA_CS_4941...	0
1	./kaggle_3m\TCGA_CS_4941_19960909\TCGA_CS_4941...	./kaggle_3m\TCGA_CS_4941_19960909\TCGA_CS_4941...	1
2	./kaggle_3m\TCGA_CS_4941_19960909\TCGA_CS_4941...	./kaggle_3m\TCGA_CS_4941_19960909\TCGA_CS_4941...	1
3	./kaggle_3m\TCGA_CS_4941_19960909\TCGA_CS_4941...	./kaggle_3m\TCGA_CS_4941_19960909\TCGA_CS_4941...	1
4	./kaggle_3m\TCGA_CS_4941_19960909\TCGA_CS_4941...	./kaggle_3m\TCGA_CS_4941_19960909\TCGA_CS_4941...	1

This is the data set we are gonna use from here. It consists of image path, which refers to the location of the Brain MRI Image and mask path, which refers to the corresponding location of the mask image path.

Now let us plot the dataset in a bar chart to divide them based on the diagnosis. The following code allows us to do that:

```python
ax = df['diagnosis'].value_counts().plot(kind='bar', stacked=True, figsize=(10,6), color=['blue', 'red'])
ax.set_title('Data Distribution')
ax.set_ylabel('Total Images', fontsize=15)
ax.set_xticklabels(['No Tumor', 'Tumor'], fontsize=12, rotation=0)
for i, rows in enumerate(df['diagnosis'].value_counts().values):
    ax.annotate(int(rows), xy=(i, rows+12), ha='center', fontweight='bold', fontsize=15)
```

The Bar chart we get is:



We can see of the 3939 image pairs in our dataset, 2556 of them have masks which indicate that they have tumor and whereas 1373 image pairs have masks that indicate otherwise. We would need both these types of image pairs as they would help us make a better rather than a biased model.

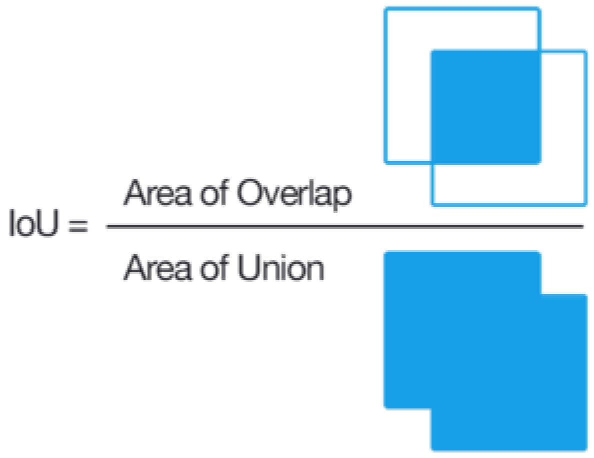
Now that we understood the model it is better to move forward to the model planning and model building.

7) Implementation of Segmentation model

Before we go into the architecture of the model itself it is better to first define a concrete loss function and metrics which would help make a better machine learning model. In case of this segmentation task we are gonna use the dice coefficient as the loss function and both iou and dice coefficent as metrics for the model.

7.a) Intersection-Over-Union (IoU, Jaccard Index)

The Intersection-Over-Union (IoU), also known as the Jaccard Index, is one of the most commonly used metrics in semantic segmentation… and for good reason. The IoU is a very straightforward metric that's extremely effective.



IoU calculation visualized. Source: Wikipedia

Before reading the following statement, take a look at the image to the left. Simply put, the **IoU is the area of overlap between the predicted segmentation and the ground truth divided by the area of union between the predicted segmentation and the ground truth**, as shown on the image to the left. This metric ranges from 0–1 (0–100%) with 0 signifying no overlap and 1 signifying perfectly overlapping segmentation.

7.b) **Dice Coefficient (F1 Score)**

Simply put, the Dice Coefficient is 2 * the Area of Overlap divided by the total number of pixels in both images. .

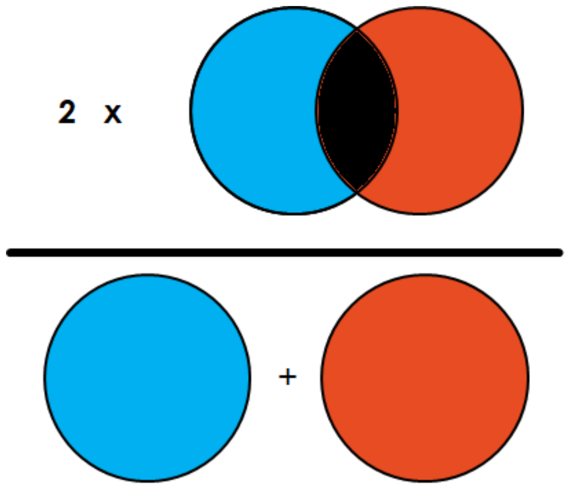
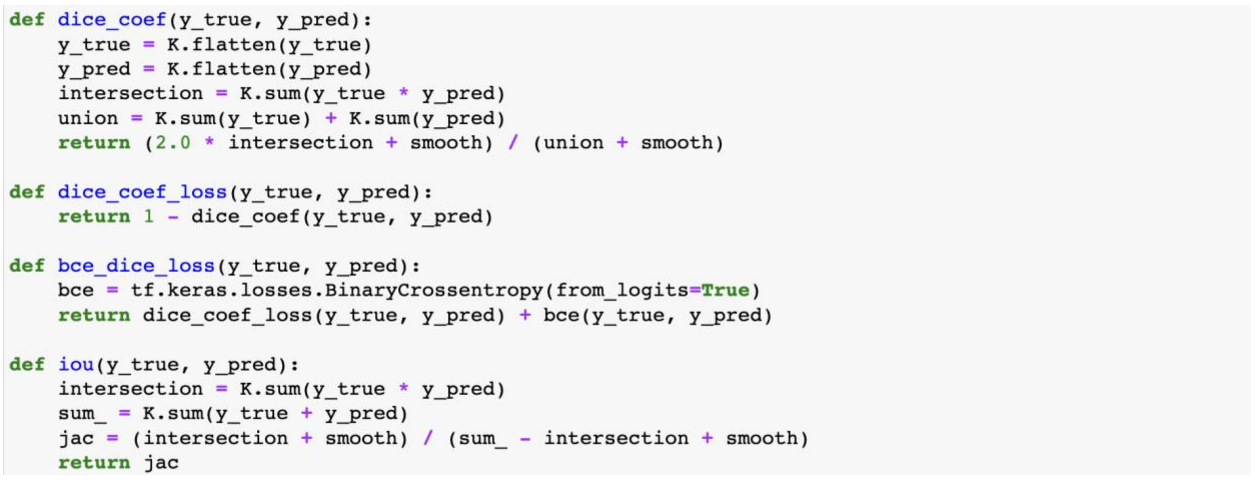


Illustration of Dice Coefficient. 2xOverlap/Total number of pixels

The Dice coefficient is very similar to the IoU. They are positively correlated, meaning if one says model A is better than model B at segmenting an image, then the other will say the same. Like the IoU, they both range from 0 to 1, with 1 signifying the greatest similarity between predicted and truth.

In the our segmentation case our metrics and loss function represent the over lap of the mask pixel in the predicted and the ground truth masks.

In the following code we are implementing the dice co-efficent and IOU co-efficent functions.



```
def dice_coef(y_true, y_pred):
    y_true = K.flatten(y_true)
    y_pred = K.flatten(y_pred)
    intersection = K.sum(y_true * y_pred)
    union = K.sum(y_true) + K.sum(y_pred)
    return (2.0 * intersection + smooth) / (union + smooth)

def dice_coef_loss(y_true, y_pred):
    return 1 - dice_coef(y_true, y_pred)

def bce_dice_loss(y_true, y_pred):
    bce = tf.keras.losses.BinaryCrossentropy(from_logits=True)
    return dice_coef_loss(y_true, y_pred) + bce(y_true, y_pred)

def iou(y_true, y_pred):
    intersection = K.sum(y_true * y_pred)
    sum_ = K.sum(y_true + y_pred)
    jac = (intersection + smooth) / (sum_ - intersection + smooth)
    return jac
```

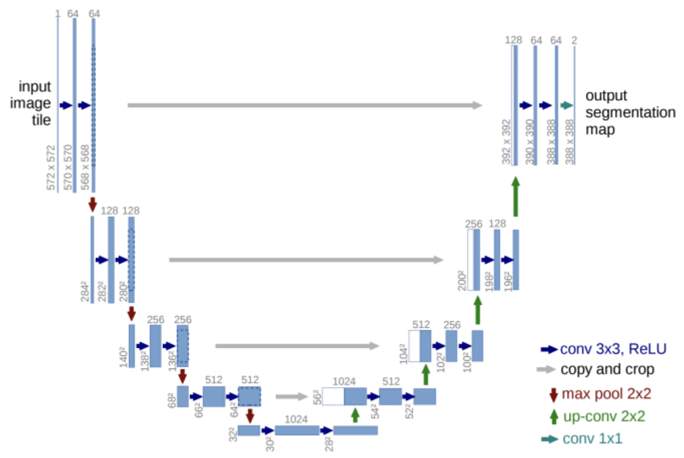
These functions are utilized during the model training phase and testing phase.

Now that we have defined the metrics for the model, let us look at the implementation of the architecture.

As we have previously seen in the introduction and background about the Image segmentation task and the working of a Convolutional neural network now we would see the implementation a U-net architecture which is widely used in the field of medical image segmantation.

7.c) U-net architecture for image segmentation[2]

The network architecture is illustrated in below Figure . It consists of a contracting path (left side) and an expansive path (right side). The contracting path follows the typical architecture of a convolutional network. It consists of the repeated application of two 3x3 convolutions (unpadded convolutions), each followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation with stride 2 for downsampling.



At each downsampling step we double the number of feature channels. Every step in the expansive path consists of an upsampling of the feature map followed by a 2x2 convolution ("up-convolution") that halves the number of feature channels, a concatenation with the correspondingly cropped feature map from the contracting path, and two 3x3 convolutions, each followed by a ReLU. The cropping is necessary due to the loss of border pixels in every convolution. At the final layer a 1x1 convolution is used to map each 64- component feature vector to the desired number of classes. In total the network has 23 convolutional layers. To allow a seamless tiling of the output segmentation map (see Figure 2), it is important to select the input tile size such that all 2x2 max-pooling operations are applied to a layer with an even x- and y-size.

The implementation of the U-net model in python is usually done with tensorflow library. This library allows us to achieve a level of abstraction in the construction of the convolution layes and the pooling layers. The unet consists of halves, The Encoder half followed by the decoder half

7.c.i) Encoder Half:
The encoder is where you apply convolution blocks followed by a maxpool

downsampling to encode the input image into feature representations at multiple different levels.

7.c.ii) Decoder Half:
The goal of this half is to semantically project the discriminative features (lower resolution) learnt by the encoder onto the pixel space (higher resolution) to get a dense classification. The decoder consists of upsampling and concatenation followed by regular convolution operations

These halves define the two most important paths of the architecture, which are the contracting paths and the expanding paths.

Contracting Path

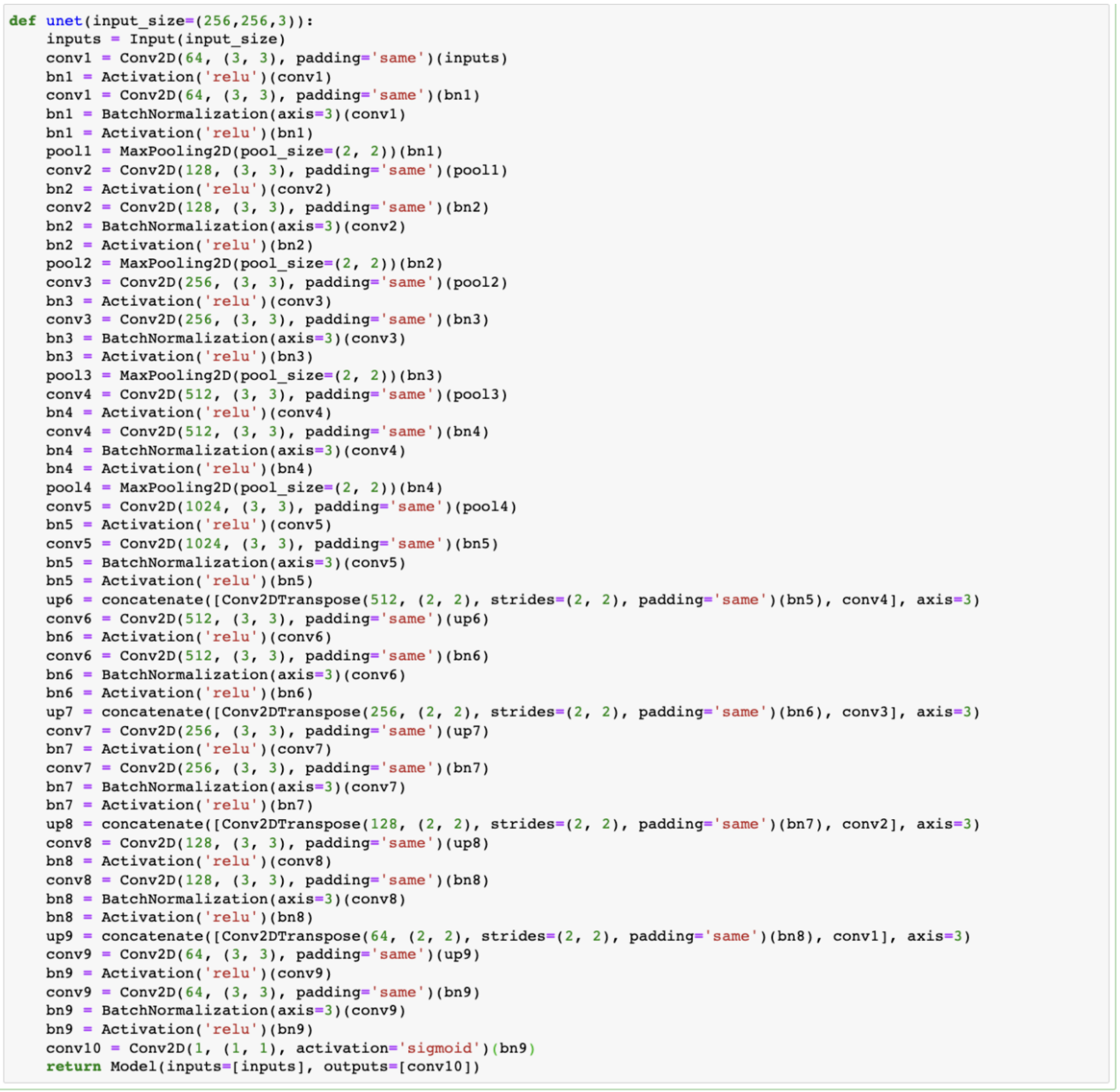
The contracting path follows the formula:

```
conv_layer1 -> conv_layer2 -> max_pooling -> dropout(optional)
```

Followed by the expanding path:
The Expanding Path follows the formula:

```
conv_2d_transpose -> concatenate -> conv_layer1 -> conv_layer2
```

In the following code we will implement both the halves and the U-net architecture in way that would fit our data set. In our dataset we convert the images in the format of (256,256) and pass on to the model.
7.d) U-net code implementation



```python
def unet(input_size=(256,256,3)):
    inputs = Input(input_size)
    conv1 = Conv2D(64, (3, 3), padding='same')(inputs)
    bn1 = Activation('relu')(conv1)
    conv1 = Conv2D(64, (3, 3), padding='same')(bn1)
    bn1 = BatchNormalization(axis=3)(conv1)
    bn1 = Activation('relu')(bn1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(bn1)
    conv2 = Conv2D(128, (3, 3), padding='same')(pool1)
    bn2 = Activation('relu')(conv2)
    conv2 = Conv2D(128, (3, 3), padding='same')(bn2)
    bn2 = BatchNormalization(axis=3)(conv2)
    bn2 = Activation('relu')(bn2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(bn2)
    conv3 = Conv2D(256, (3, 3), padding='same')(pool2)
    bn3 = Activation('relu')(conv3)
    conv3 = Conv2D(256, (3, 3), padding='same')(bn3)
    bn3 = BatchNormalization(axis=3)(conv3)
    bn3 = Activation('relu')(bn3)
    pool3 = MaxPooling2D(pool_size=(2, 2))(bn3)
    conv4 = Conv2D(512, (3, 3), padding='same')(pool3)
    bn4 = Activation('relu')(conv4)
    conv4 = Conv2D(512, (3, 3), padding='same')(bn4)
    bn4 = BatchNormalization(axis=3)(conv4)
    bn4 = Activation('relu')(bn4)
    pool4 = MaxPooling2D(pool_size=(2, 2))(bn4)
    conv5 = Conv2D(1024, (3, 3), padding='same')(pool4)
    bn5 = Activation('relu')(conv5)
    conv5 = Conv2D(1024, (3, 3), padding='same')(bn5)
    bn5 = BatchNormalization(axis=3)(conv5)
    bn5 = Activation('relu')(bn5)
    up6 = concatenate([Conv2DTranspose(512, (2, 2), strides=(2, 2), padding='same')(bn5), conv4], axis=3)
    conv6 = Conv2D(512, (3, 3), padding='same')(up6)
    bn6 = Activation('relu')(conv6)
    conv6 = Conv2D(512, (3, 3), padding='same')(bn6)
    bn6 = BatchNormalization(axis=3)(conv6)
    bn6 = Activation('relu')(bn6)
    up7 = concatenate([Conv2DTranspose(256, (2, 2), strides=(2, 2), padding='same')(bn6), conv3], axis=3)
    conv7 = Conv2D(256, (3, 3), padding='same')(up7)
    bn7 = Activation('relu')(conv7)
    conv7 = Conv2D(256, (3, 3), padding='same')(bn7)
    bn7 = BatchNormalization(axis=3)(conv7)
    bn7 = Activation('relu')(bn7)
    up8 = concatenate([Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='same')(bn7), conv2], axis=3)
    conv8 = Conv2D(128, (3, 3), padding='same')(up8)
    bn8 = Activation('relu')(conv8)
    conv8 = Conv2D(128, (3, 3), padding='same')(bn8)
    bn8 = BatchNormalization(axis=3)(conv8)
    bn8 = Activation('relu')(bn8)
    up9 = concatenate([Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same')(bn8), conv1], axis=3)
    conv9 = Conv2D(64, (3, 3), padding='same')(up9)
    bn9 = Activation('relu')(conv9)
    conv9 = Conv2D(64, (3, 3), padding='same')(bn9)
    bn9 = BatchNormalization(axis=3)(conv9)
    bn9 = Activation('relu')(bn9)
    conv10 = Conv2D(1, (1, 1), activation='sigmoid')(bn9)
    return Model(inputs=[inputs], outputs=[conv10])
```

We have taken a relu activation function. This activation function is used in the final part of the model to output the encoded mask pixel values.

Now that we are able to implement the code of the model let continue with the model training and testing. This can be followed by the predictions.

8) Training:

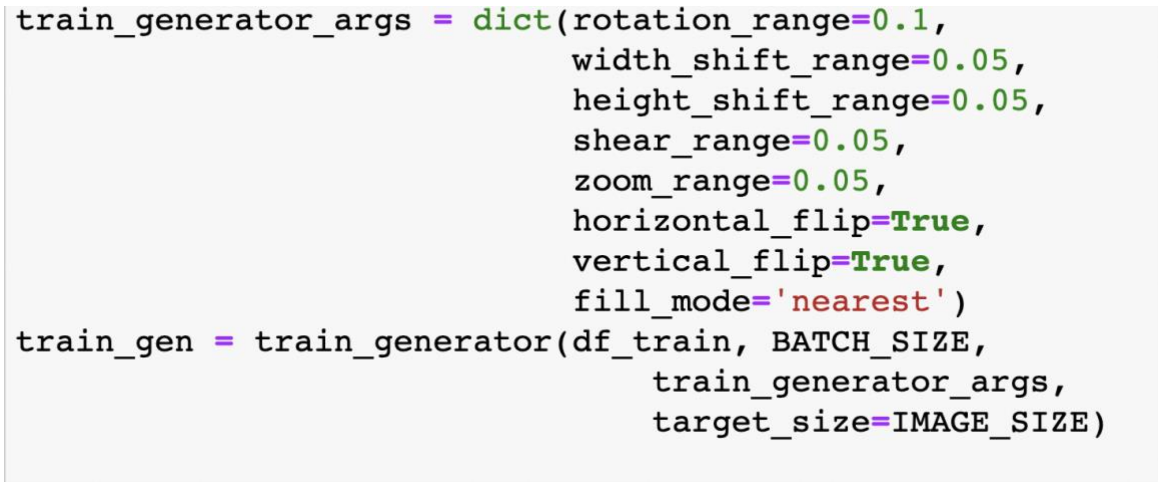
To Train the model we would create a train_generator function . This helps us automate the task of feeding the dataset in to the Model. The tensorflow.keras.preprocessing.image library provides us with all the required functions. Apart from constructing a image feeder we have to create a generator.

As we have only 3929 images which is little we need to create a data generator with the help of data augmentation

8.a) Data Augmentation:

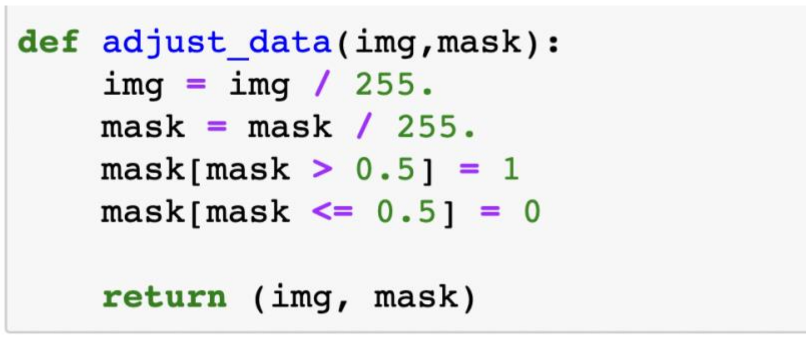
Data augmentation is essential to teach the network the desired invariance and robustness properties, when only few training samples are available. In case of Brain MRI images we primarily need shift and rotation invariance as well as robustness to deformations and gray value variations. Especially random elastic deformations of the training samples seem to be the key concept to train a segmentation network with very few annotated images. We generate smooth deformations using random displacement vectors on a coarse 3 by 3 grid. The displacements are sampled from a Gaussian distribution with 10 pixels standard deviation. Per-pixel displacements are then computed using bicubic interpolation. Drop-out layers at the end of the contracting path perform further implicit data augmentation.

All these calculations can be abstracted as the keras library function implement them for us.



```python
train_generator_args = dict(rotation_range=0.1,
                            width_shift_range=0.05,
                            height_shift_range=0.05,
                            shear_range=0.05,
                            zoom_range=0.05,
                            horizontal_flip=True,
                            vertical_flip=True,
                            fill_mode='nearest')
train_gen = train_generator(df_train, BATCH_SIZE,
                                train_generator_args,
                                target_size=IMAGE_SIZE)
```

We also include the adjust data function which helps us in smoothing out the pixel data values in the mask and round them of for the convenience of the model.



```python
def adjust_data(img,mask):
    img = img / 255.
    mask = mask / 255.
    mask[mask > 0.5] = 1
    mask[mask <= 0.5] = 0

    return (img, mask)
```

8.b) Training

The training is done using model.fit function call, where we initiate the Unet architecture as the model.

Before we run the model.fit function we try to implement callback criterias and also try to define model check points.

```
model = unet(input_size=(IMAGE_SIZE[0], IMAGE_SIZE[1], 3))


opt = Adam(lr=learning_rate, beta_1=0.9, beta_2=0.999, epsilon=None, amsgrad=False)
model.compile(optimizer=opt, loss=bce_dice_loss, metrics=[iou, dice_coef])

callbacks = [ModelCheckpoint('unet_brainMRI_seg.hdf5', verbose=0, save_best_only=True),
            ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=5, verbose=1, min_lr=1e-11),
            EarlyStopping(monitor='val_loss', restore_best_weights=True, patience=15)]
```

We use an Adam optimizer in the training of the model and we use the bce-dice-loss function which utiliszes the dice co-efficent which we have discussed earlier.

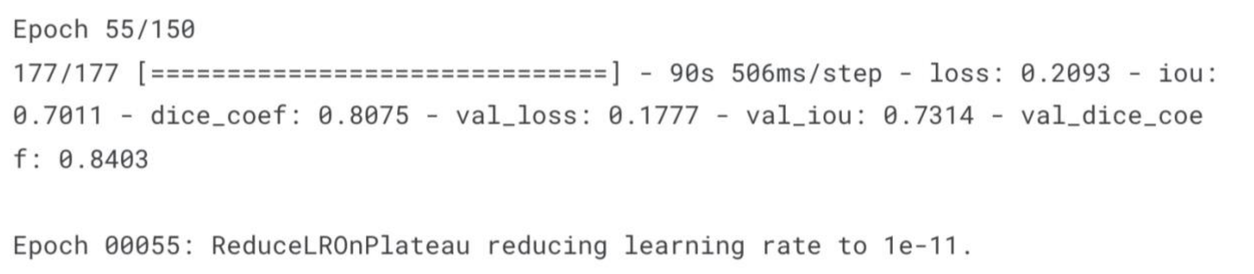
Now that we have defined all the required parameters for the model we train the model.

```
history = model.fit(train_gen,
                    steps_per_epoch=len(df_train) / BATCH_SIZE,
                    epochs=EPOCHS,
                    callbacks=callbacks,
                    validation_data = val_gen,
                    validation_steps=len(df_val) / BATCH_SIZE)
```

We run the model for 150 epochs with a batch size of 16. During the run time we get updated messages in the console stating the dice co-efficents and various other parameters at each epoch.

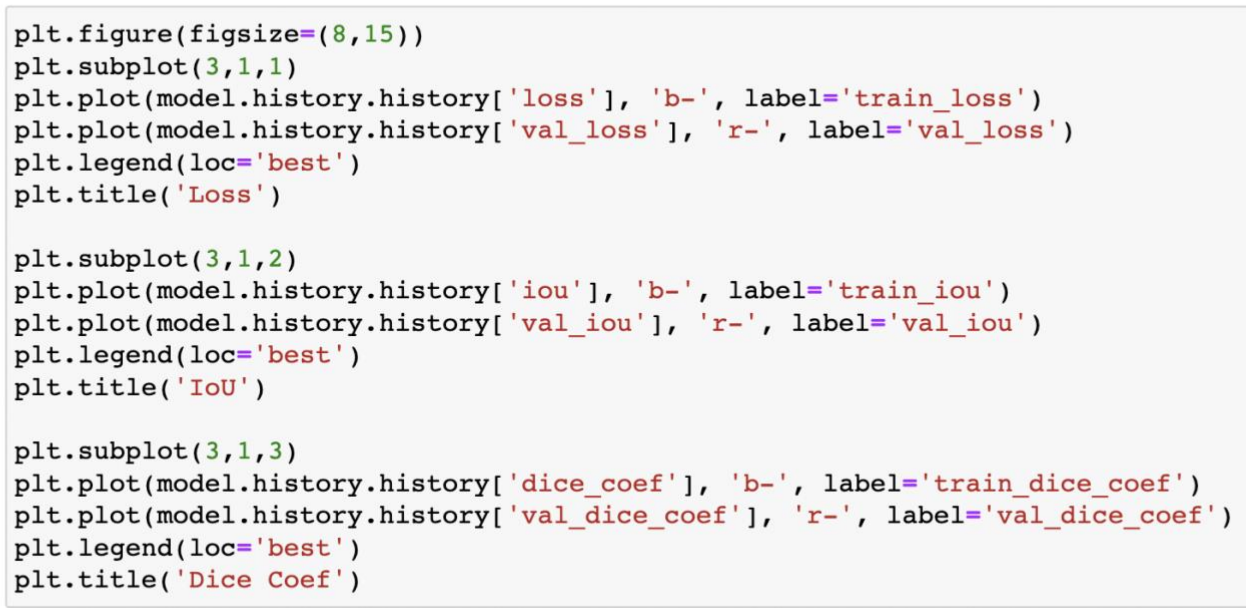
```
Found 2838 validated image filenames.
Found 2838 validated image filenames.
Epoch 1/150
178/177 [==============================] - ETA: 0s - loss: 1.1957 - iou: 0.0384
- dice_coef: 0.0725Found 501 validated image filenames.
Found 501 validated image filenames.
177/177 [==============================] - 116s 582ms/step - loss: 1.1950 - io
u: 0.0385 - dice_coef: 0.0727 - val_loss: 1.2594 - val_iou: 0.0131 - val_dice_c
oef: 0.0258
Epoch 2/150
177/177 [==============================] - 88s 495ms/step - loss: 0.9062 - iou:
0.0993 - dice_coef: 0.1777 - val_loss: 1.0126 - val_iou: 0.0356 - val_dice_coe
f: 0.0673
```

Our respective model plateaus at 55 epochs so the learning stops and we reach the end of the training process. This is the final message outputted at the console at the end of training.



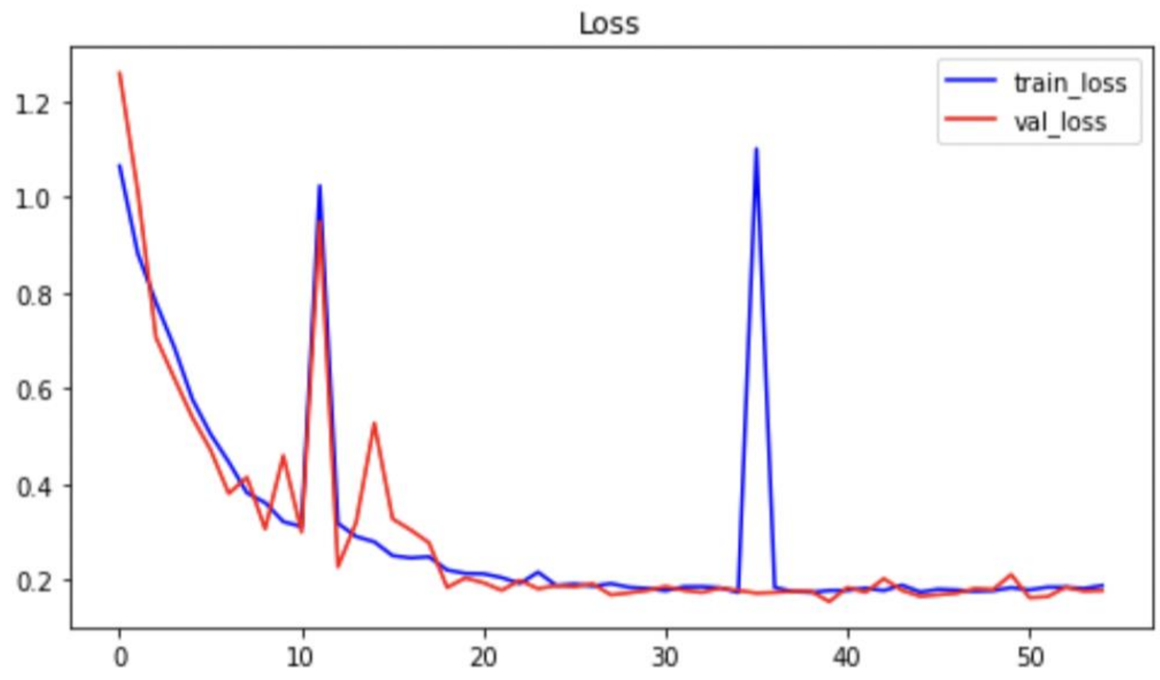
```
Epoch 55/150
177/177 [==============================] - 90s 506ms/step - loss: 0.2093 - iou:
0.7011 - dice_coef: 0.8075 - val_loss: 0.1777 - val_iou: 0.7314 - val_dice_coe
f: 0.8403

Epoch 00055: ReduceLROnPlateau reducing learning rate to 1e-11.
```

As the model has concluded training we can visualize the models performance by the use of a graph which can shows us how the loss has changed over the training period. The following code allows us to visualize the loss functions.

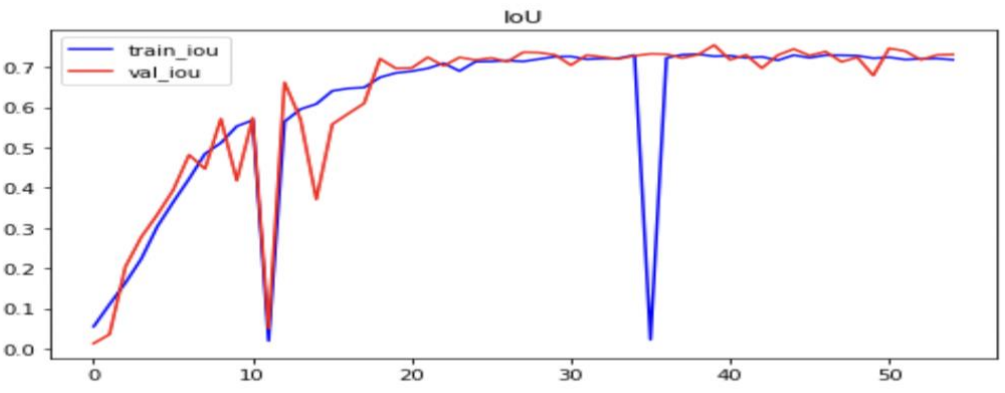


```
plt.figure(figsize=(8,15))
plt.subplot(3,1,1)
plt.plot(model.history.history['loss'], 'b-', label='train_loss')
plt.plot(model.history.history['val_loss'], 'r-', label='val_loss')
plt.legend(loc='best')
plt.title('Loss')

plt.subplot(3,1,2)
plt.plot(model.history.history['iou'], 'b-', label='train_iou')
plt.plot(model.history.history['val_iou'], 'r-', label='val_iou')
plt.legend(loc='best')
plt.title('IoU')

plt.subplot(3,1,3)
plt.plot(model.history.history['dice_coef'], 'b-', label='train_dice_coef')
plt.plot(model.history.history['val_dice_coef'], 'r-', label='val_dice_coef')
plt.legend(loc='best')
plt.title('Dice Coef')
```

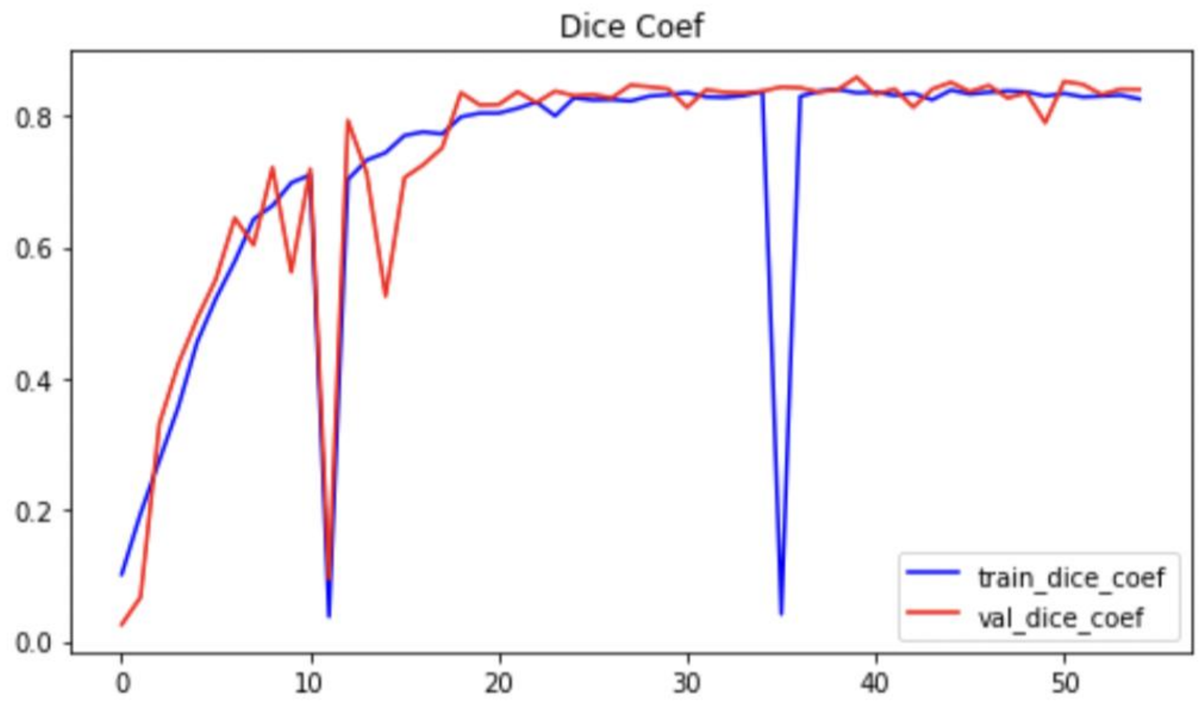
The graphs plotted by plt are as follows:



This is the loss graph. It shows both the training loss as well as the validation loss during the training process. The x-axis represent the epochs hence the graph stops at 55. Starting at 1.19 loss values we ended up at 0.29. Even the loss graph help us understand that the model is learning with each epoch an Iou and Dice co-efficent graph help us better understand how accurate the models is.
The IOU and the Dice co-efficent graphs are as followed respectively:



As we saw earlier that the better the IOU the better the model is so as our model has almost an IOU value of 73 our model has performed fairly well. Now we take a look at the Dice co-efficent graph as it is the main graph which helps us understand how good the model is. The dice co-efficent graph is as follows



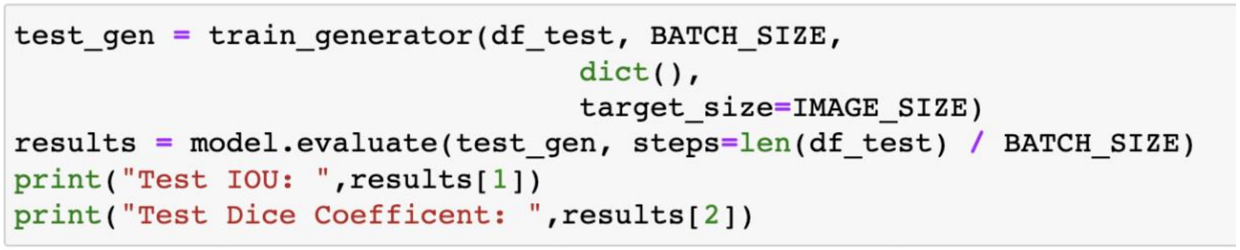
We can see that the model has a dice co-efficent of 0.83 which means it has an accuracy of 83% which is a decent value given that we had only a little amount of data.

Hence as we have completed the training we can focus on the results and further try to test the model on random samples of brain MRI Images. This not only gives us a concrete evidence on the accuracy of the model but also a non-technical representation of out model outcomes

9) Model Testing and results:

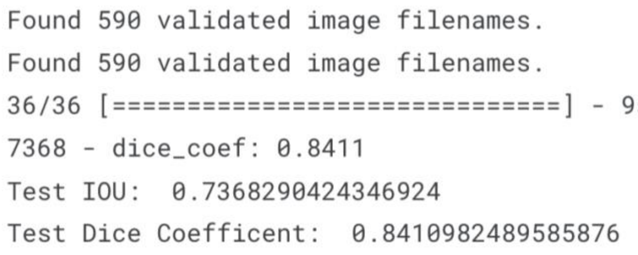
We try to test the model by running the model of the df_test dataset, which is the test part of the dataset we have partitioned before the testing process has started. By running on the the test dataset we would be able to check the true accuracy of the model and seee if any case of overfitting or underfitting has taken place which would make us to retrain the model using different set of parameters.

The following code helps us test the data set on the test dataset. We use the model.evalute function which returns us the accuracy with the dice co-efficents and respective IOU co-efficents.



```
test_gen = train_generator(df_test, BATCH_SIZE,
                                dict(),
                                target_size=IMAGE_SIZE)
results = model.evaluate(test_gen, steps=len(df_test) / BATCH_SIZE)
print("Test IOU: ",results[1])
print("Test Dice Coefficent: ",results[2])
```

The final results of testing displayed on the output are:



```
Found 590 validated image filenames.
Found 590 validated image filenames.
36/36 [==============================] - 9s 247ms/step - loss: 0.1712 - iou: 0.
7368 - dice_coef: 0.8411
Test IOU:  0.7368290424346924
Test Dice Coefficent:  0.8410982489585876
```

So, on the test data we are able to reach a loss of 0.1712. Even though such low loss is a good indicator that our model is working properly. A better indicator is the Dice and IOU co-efficents which help us visualize. In this case the Test_IOU is 0.73 which stands for 73% accuracy using the IOU co-efficent and the Test Dice

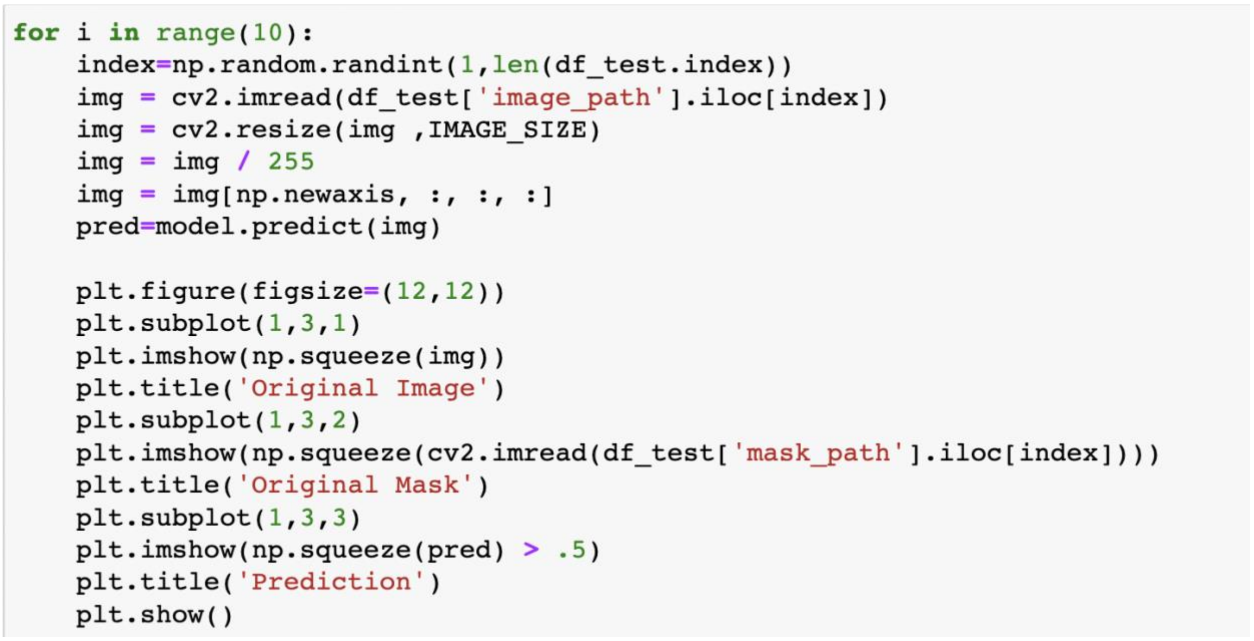
Coefficent is 0.84 which stands for 84%. Both these values indicate that out model has achieved its required goal without any bias as our dataset consists of image-mask pairs which are different that those is the training phase.

9.a) Visualizing the results.

Now we can try to visualize the results by displaying the Brain MRI Image, the corresponding ground truth mask and the predicted mask.

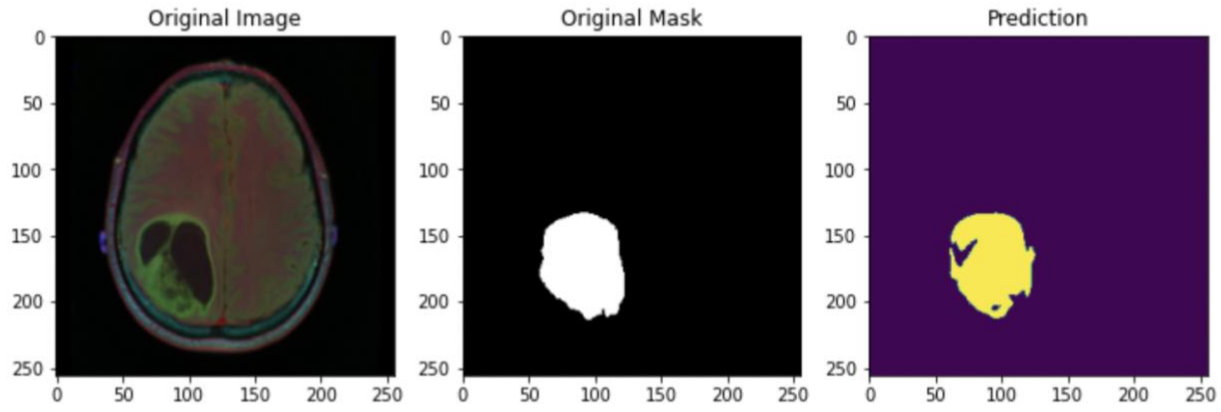
We take the test dataset and randomly select the ground truth image mask pairs and then pass on the MRI images into the model and try to see the outputs.

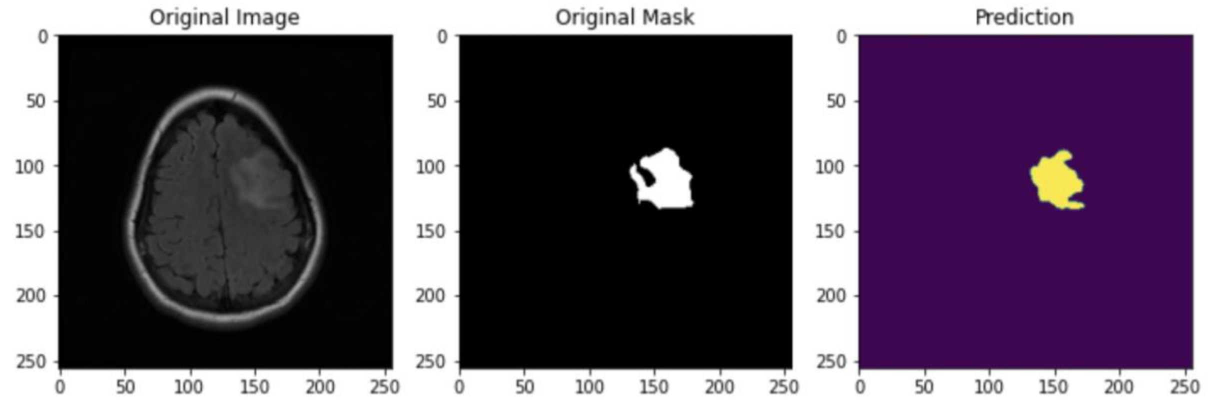
The following code consists of the operation of prediction and plotting all three elements of output ion a row wise order in order to better visualize the results.

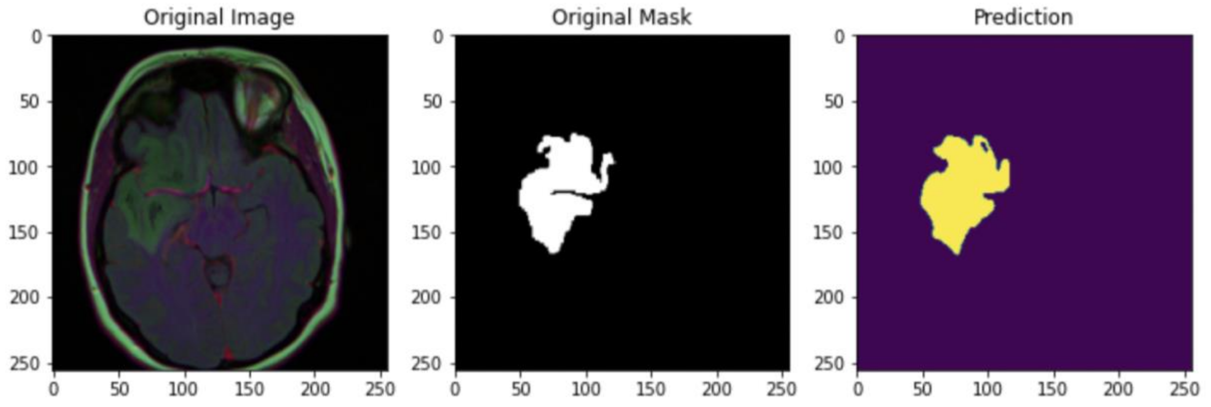


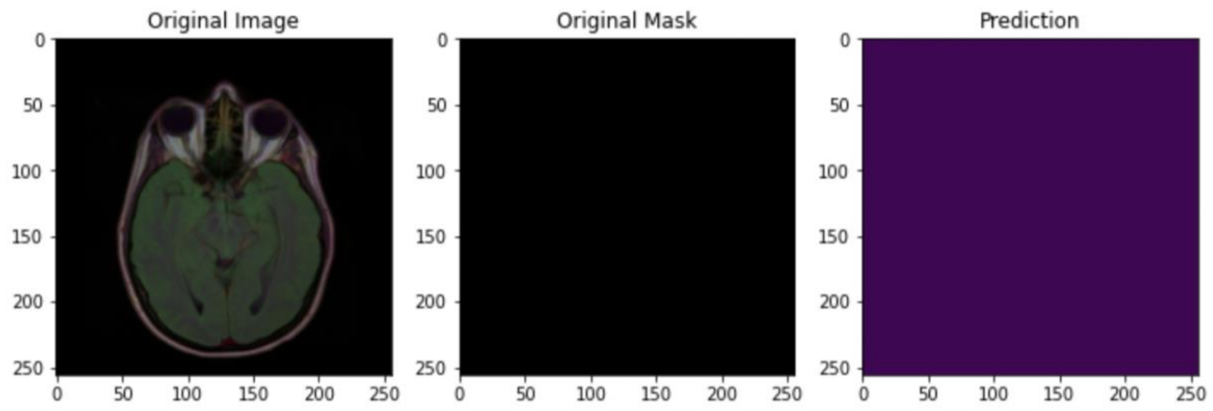
```
for i in range(10):
    index=np.random.randint(1,len(df_test.index))
    img = cv2.imread(df_test['image_path'].iloc[index])
    img = cv2.resize(img ,IMAGE_SIZE)
    img = img / 255
    img = img[np.newaxis, :, :, :]
    pred=model.predict(img)

    plt.figure(figsize=(12,12))
    plt.subplot(1,3,1)
    plt.imshow(np.squeeze(img))
    plt.title('Original Image')
    plt.subplot(1,3,2)
    plt.imshow(np.squeeze(cv2.imread(df_test['mask_path'].iloc[index])))
    plt.title('Original Mask')
    plt.subplot(1,3,3)
    plt.imshow(np.squeeze(pred) > .5)
    plt.title('Prediction')
    plt.show()
```

A few of the predictions are as displayed:


Original Image
Original Mask
Prediction
0
50
100
150
200
250


Original Image
Original Mask
Prediction
0
50
100
150
200
250


Original Image
Original Mask
Prediction
0
50
100
150
200
250



Hence we can see from the visualizations that our model is able to predict masks with a fair certain of accuracy. As we have completed all the important tasks of the machine learning model i.e training and testing and also visualizing the results we can conclude the project.

Conclusion(and further improvements):

As we have discussed in the start of the report our main goal is to implement a robust and useful machine learning model for Brain trumor segmentation. We were able to create a decent Image segmentation model using existing methodologies which gave us good i.e 83% accuracy. We were even able to visualize the results and understand the good our model is. Even though 84% accuracy is respectable given our dataset size, further improvements can be made to the project by taking a better architectured CNN which can be tailored for out dataset for example a resnet. With these statement we can conclude the project and the project report

-----------------------------X---------------------------------

References

1. Maciej A. Mazurowski, Kal Clark, Nicholas M. Czarnek, Parisa Shamsesfandabadi, Katherine B. Peters, Ashirbani Saha "Radiogenomics of lower-grade glioma: algorithmically-assessed tumor shape is associated with tumor genomic subtypes and patient outcomes in a multi-institutional study with The Cancer Genome Atlas data." Journal of Neuro-Oncology, 2017.
2. U-NET: Convolutional Networks for Biomedical Image Segmentation, Olaf Ronneberger, Philipp Ficher, Thomas Brox, Computer Vision and Pattern Recognition(cs,CV)