

Brain Tumor Classification

Brain Magnetic Resonance Images

Team Members:

Sanika Mhadgut - J031

Gayathri Shrikanth - J046

Our Aim:

Classifying Brain MRI Scans based on existence of tumors.

Detect and highlight the tumor in the Image.

Introduction:

Magnetic resonance imaging (MRI) is the most common imaging technique used to detect abnormal brain tumors. Traditionally, MRI images are analyzed manually by radiologists to detect the abnormal conditions in the brain. Manual interpretation of huge volumes of images is time consuming and difficult. Hence, computer-based detection helps in accurate and fast diagnosis. In this study, we proposed an approach that uses deep transfer learning to automatically classify normal and abnormal brain MR images.

The Dataset:

Brain MRI Images for Brain Tumor Detection.

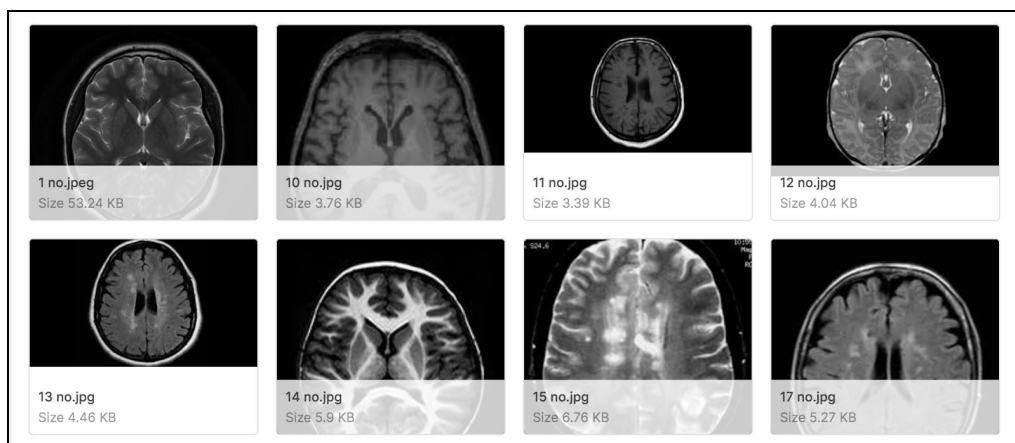


Figure 2. Sample normal brain MR images used.

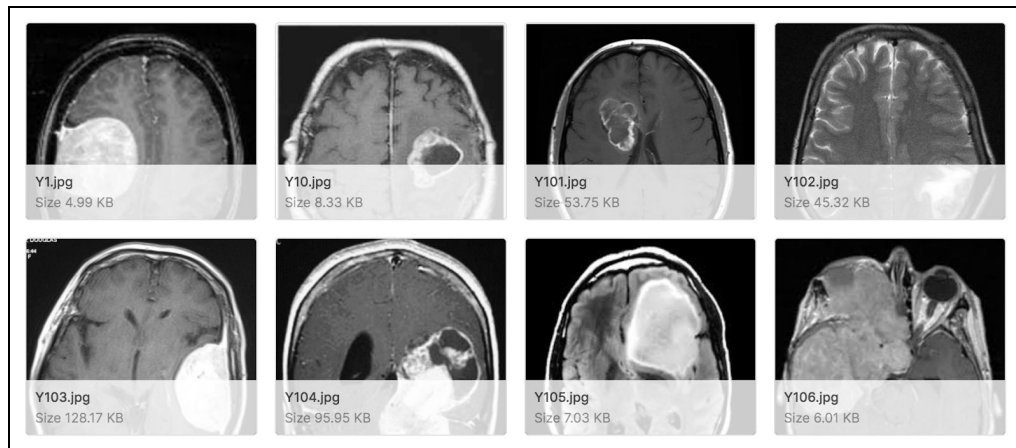


Figure 3. Sample abnormal brain MR images used.

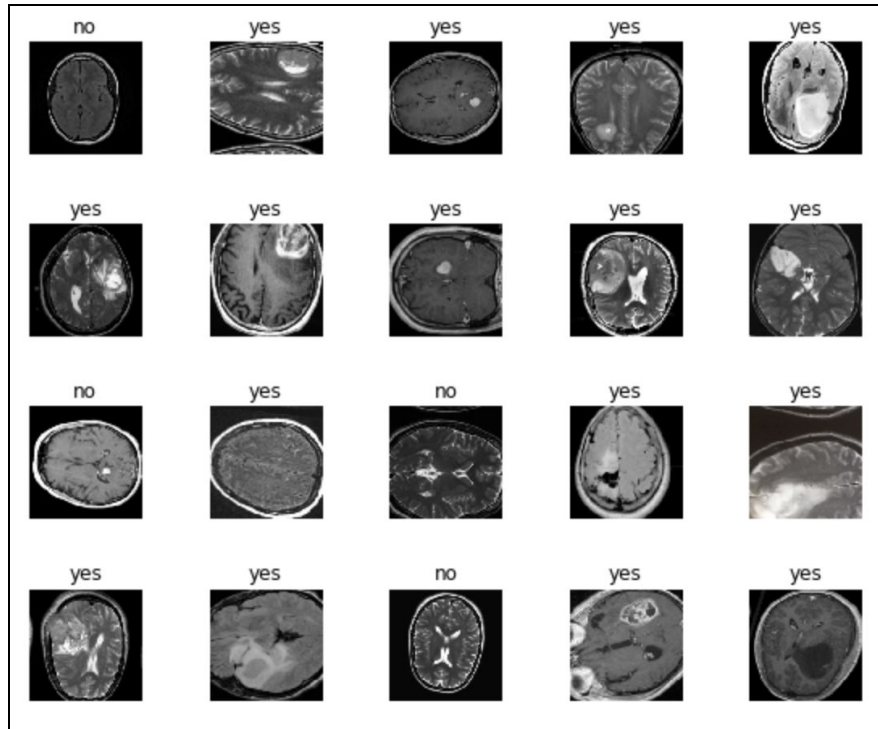
Part 1: Brain Tumor Classification using Fast.ai

Step 1: Inspecting the Dataset

A total of 253 images were used for training and validations. Among these images, 98 were normal images and rest 155 were abnormal images.

```
No. of labels: 2
-----
yes, 155 files
no, 98 files
```

Figure 1. Dataset



Step 2: Reading data into the model

Since the images were in folders, we used `ImageDataBunch.from_folder()` function to create an object that contains our image data.

```
data = ImageDataBunch.from_folder(DATA_DIR, train=".",
                                  valid_pct=0.25,
                                  ds_tfms=get_transforms(do_flip=True, flip_vert=True),
                                  size=224, bs=24,
                                  num_workers=0).normalize(imagenet_stats)
print(f'Classes: \n {data.classes}')
```

The parameters specified :

- `ds_tfms` - the transforms to apply to the images
Do_flip, flip_vert = True, because want the images flipped, vertically
- `valid_pct` - to split the test train in 25:75 ratio, controls the percentage of images that will be randomly chosen to be in the validation set

- Size = 224, the ImageDataBunch object will scale all images to a size*size squared image; bigger the image size means more details the CNN will be able to extract, but along with that computation will take longer time.
- bs = 24, batch size, if batch size is too large, it will cause the GPU to run out of memory
- Normalize - Since we will be using a ResNet, VGG architecture for our model which was trained on ImageNet, we will be using the ImageNet stats. In order to ensure that the input images have the same characteristics we normalize them. This also helps make computation faster.

Step 3: Creating FastAI Model

```
learner = create_cnn(data, models.vgg16, metrics=[accuracy],  
                    callback_fns=ShowGraph, model_dir="/tmp/model/")
```

This function creates a learner object. We specified the VGG architecture as our base model for transfer learning. Upon call, the trained architecture was downloaded via the Fastai API and stored locally.

The parameters specified :

- Metrics - We used accuracy for our metric.
- Callback_fns - The callback function ShowGraph simply tells the learner that it should return a graph for each action, which was usefly for us to check if the model was improving.

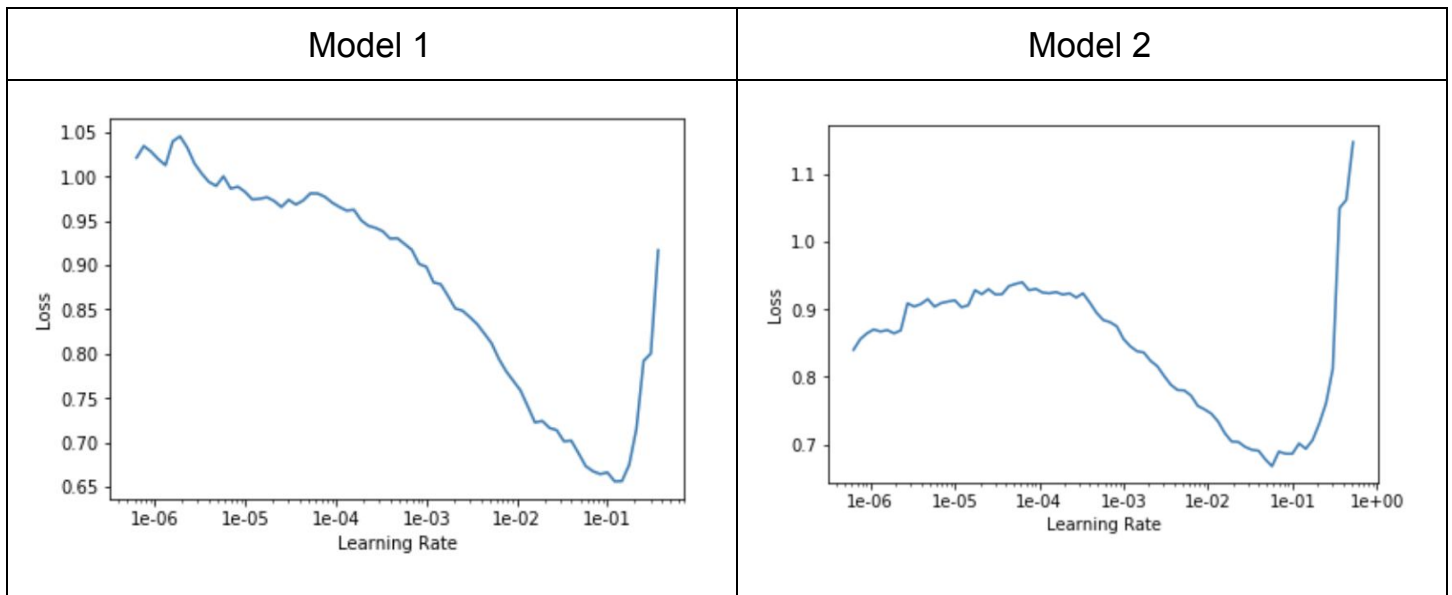
Step 4: Finding the Learning Rate [Stage 1 : Before Unfreeze]

```
learner.lr_find()  
learner.recorder.plot()
```

The learner object we created has a built-in function to find the optimal learning rate, or range of learning rates, for training. It achieves this by fitting the model for a few epochs and saving for which learning rates the loss decreases the most.

We choose a learning rate, for which the loss is still decreasing, a rate with the steepest slope.

In the following plot, which is stored in the recorder object of our learner, we can see that the slope is decreasing in between $e-02$ and $e-01$ (0.02 and 0.01).



Step 5: Fitting first model [Stage 1 : Before Unfreeze]

We fit our VGG16 model with a learning rate of 0.02 for each model.

```
learner.fit_one_cycle(15, max_lr=slice(1e-2))
```

Model 1

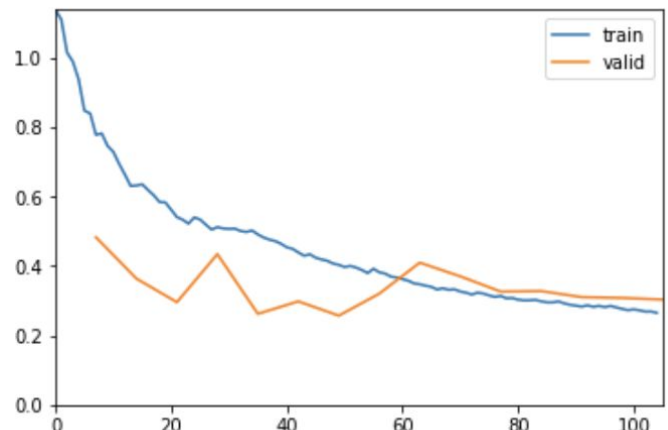
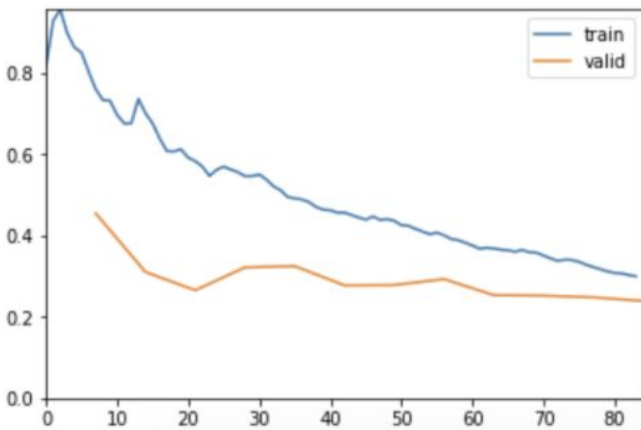
```
learner.fit_one_cycle(12, max_lr=slice(1e-02))
```

epoch	train_loss	valid_loss	accuracy	time
0	0.802675	0.454149	0.773333	00:03
1	0.735773	0.310170	0.866667	00:03
2	0.591421	0.265334	0.906667	00:03
3	0.555581	0.321732	0.880000	00:03
4	0.494803	0.324660	0.880000	00:03
5	0.456317	0.277358	0.866667	00:03
6	0.440347	0.278215	0.866667	00:03
7	0.407057	0.292924	0.866667	00:04
8	0.369834	0.253235	0.893333	00:03
9	0.357763	0.252117	0.906667	00:03
10	0.328064	0.247893	0.906667	00:04
11	0.299622	0.239090	0.906667	00:04

Model 2

```
learner.fit_one_cycle(15, max_lr=slice(1e-2))
```

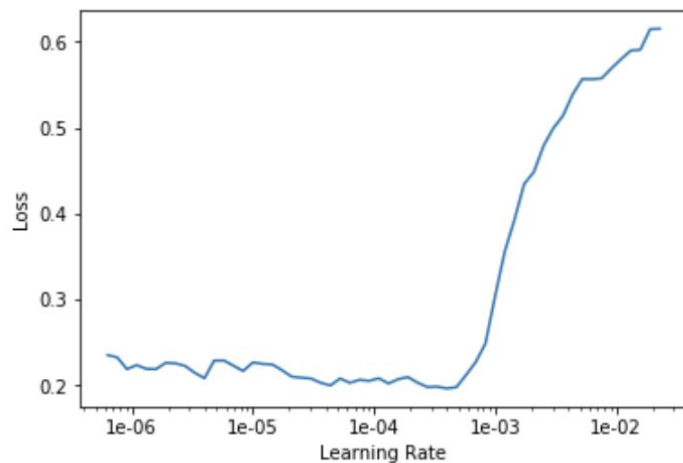
epoch	train_loss	valid_loss	accuracy	time
0	0.840369	0.483324	0.793651	01:30
1	0.631246	0.364055	0.873016	01:21
2	0.562968	0.296126	0.888889	01:20
3	0.505547	0.434782	0.873016	01:20
4	0.502834	0.262470	0.857143	01:19
5	0.449856	0.298428	0.857143	01:20
6	0.407626	0.257222	0.873016	01:20
7	0.392351	0.320714	0.873016	01:19
8	0.350323	0.410134	0.857143	01:30
9	0.333141	0.370632	0.825397	01:19
10	0.311230	0.326670	0.873016	01:21
11	0.302691	0.328102	0.873016	01:19
12	0.286109	0.310679	0.888889	01:19
13	0.280957	0.308322	0.873016	01:21
14	0.265318	0.303820	0.888889	01:20



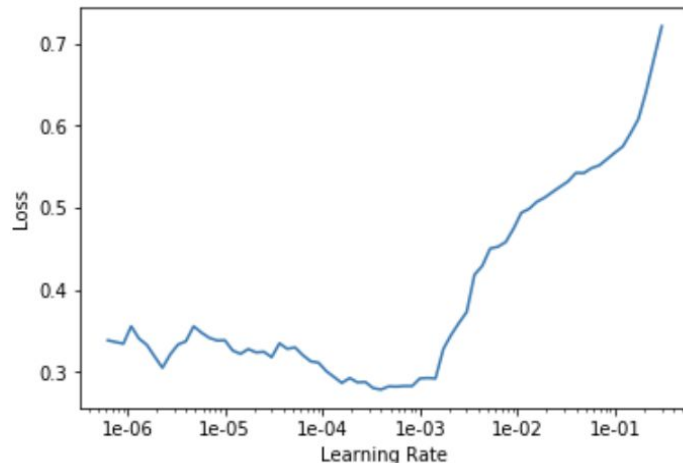
Step 6: Unfreezing and Fine-tuning to Improving the Model

```
learner.unfreeze()
learner.fit_one_cycle(10, max_lr=slice(1e-05))
```

Model 1



Model 2

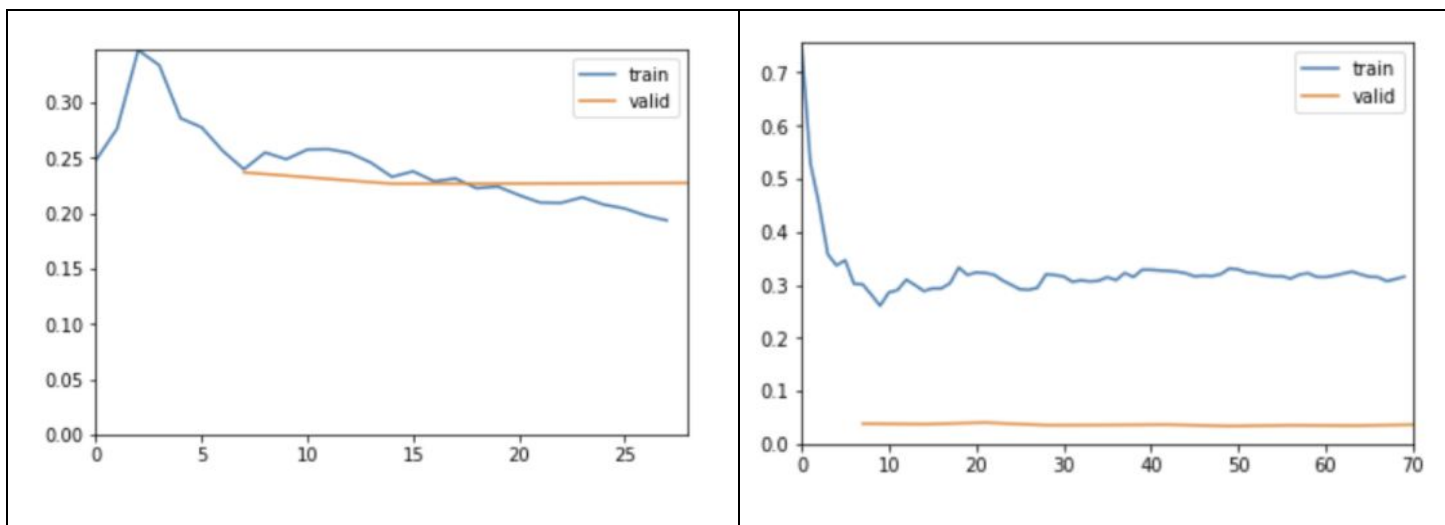


```
learner.fit_one_cycle(4, max_lr=slice(1e-04))
```

epoch	train_loss	valid_loss	accuracy	time
0	0.256503	0.237029	0.920000	00:04
1	0.245899	0.226825	0.920000	00:04
2	0.216539	0.227001	0.920000	00:04
3	0.193629	0.227553	0.933333	00:04

```
learner.fit_one_cycle(10, max_lr=slice(1e-05))
```

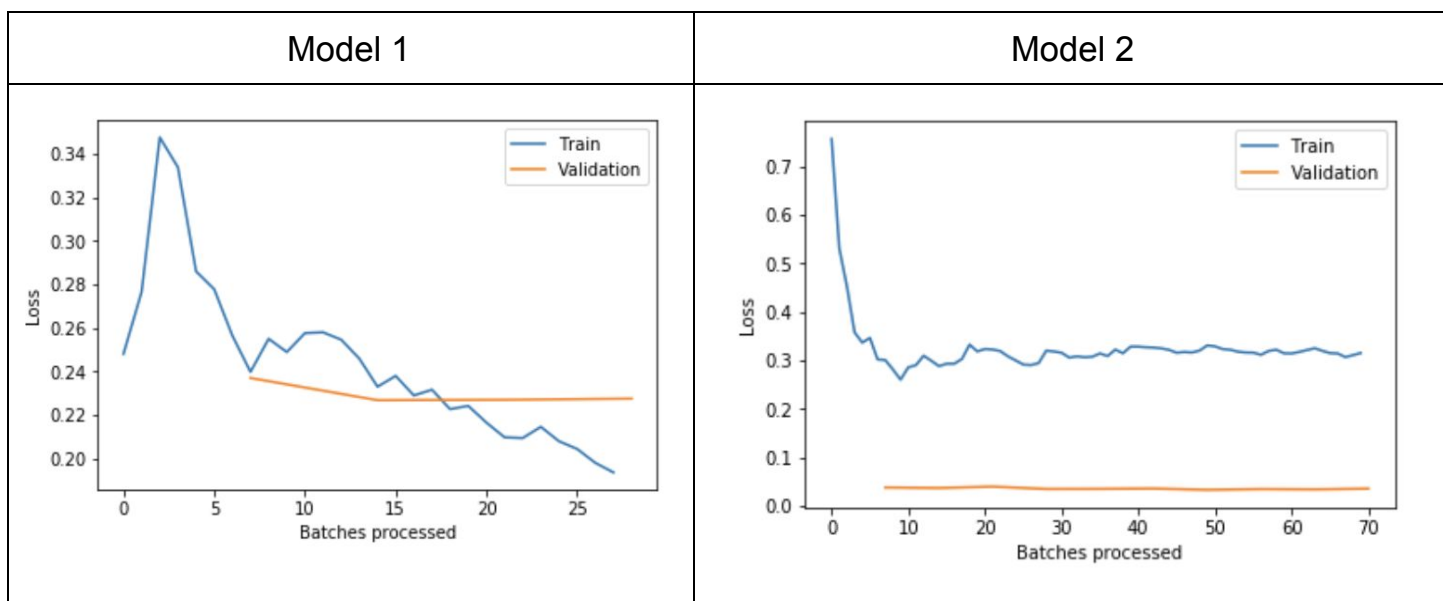
epoch	train_loss	valid_loss	accuracy	time
0	0.302036	0.038451	1.000000	03:04
1	0.299327	0.037346	1.000000	03:01
2	0.323480	0.040194	0.984127	03:16
3	0.294442	0.035387	1.000000	03:05
4	0.307581	0.035635	0.984127	03:04
5	0.327095	0.036309	0.984127	03:02
6	0.320274	0.033641	1.000000	03:02
7	0.316040	0.035161	0.984127	03:02
8	0.321303	0.034549	1.000000	03:04
9	0.315197	0.036138	1.000000	03:02



The model is overfitting when the valid loss is more than the training loss.

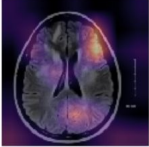
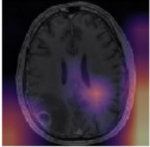
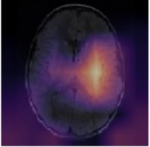
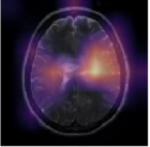
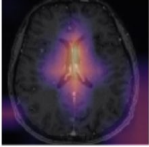
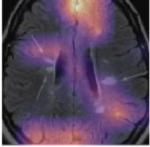
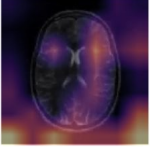
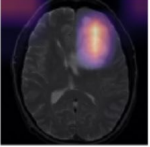
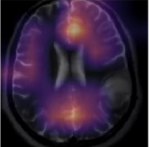
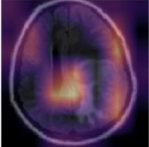
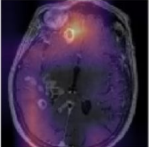
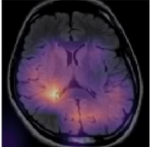
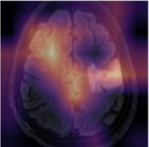
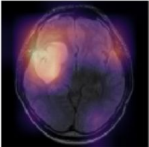
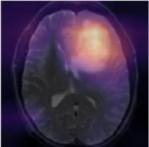
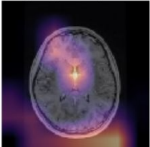
Step 7: Plotting losses

```
learner.recorder.plot_losses()
```



Step 8: Plotting top losses

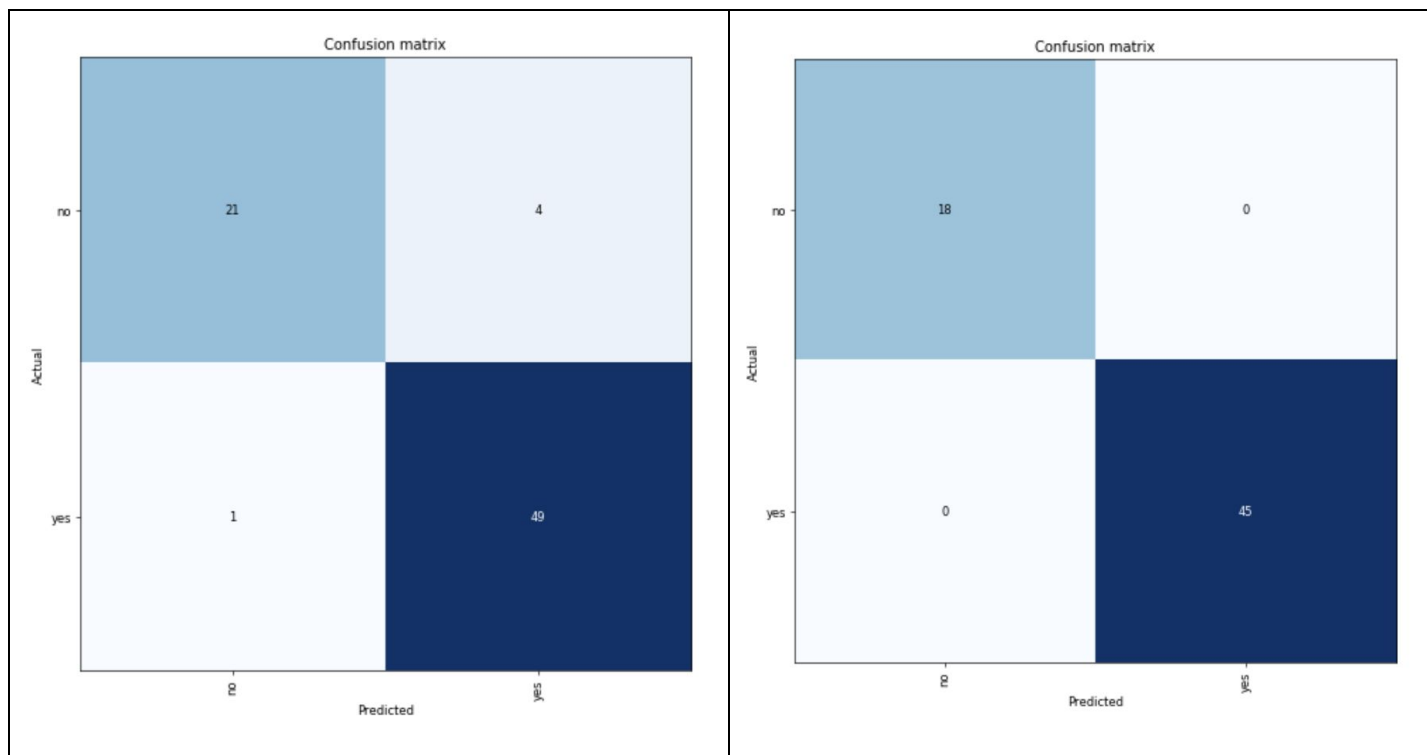
```
interp.plot_top_losses(10, figsize=(10,10))
```


Model 1	Model 2
<p>prediction/actual/loss/probability</p> <div><div>yes/no / 4.05 / 0.02</div><div></div><div>yes/no / 2.91 / 0.05</div><div></div><div>yes/no / 2.54 / 0.08</div><div></div><div>no/yes / 1.36 / 0.26</div><div></div></div> <div><div>yes/no / 1.15 / 0.32</div><div></div><div>no/no / 0.68 / 0.51</div><div></div><div>no/no / 0.67 / 0.51</div><div></div><div>yes/yes / 0.62 / 0.54</div><div></div></div>	<p>prediction/actual/loss/probability</p> <div><div>no/no / 0.65 / 0.52</div><div></div><div>yes/yes / 0.55 / 0.58</div><div></div><div>yes/yes / 0.22 / 0.81</div><div></div><div>no/no / 0.12 / 0.89</div><div></div></div> <div><div>yes/yes / 0.09 / 0.91</div><div></div><div>yes/yes / 0.08 / 0.92</div><div></div><div>yes/yes / 0.07 / 0.94</div><div></div><div>no/no / 0.05 / 0.95</div><div></div></div>

Step 9: Plotting Confusion Matrix

```
interp.plot_confusion_matrix(figsize=(8,8), dpi=60)
```

Model 1	Model 2
---------	---------



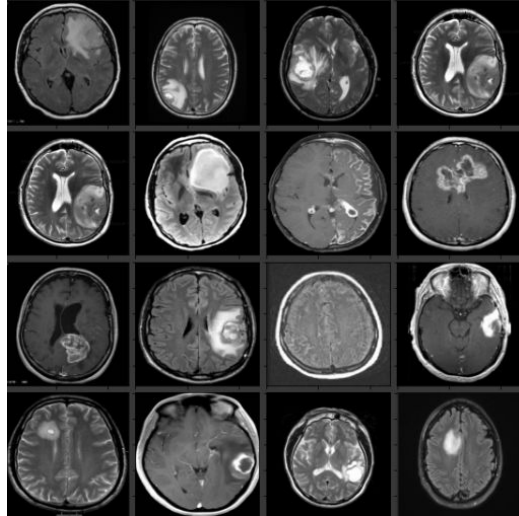
We successfully trained two models, achieving 93.33% and 100% accuracy respectively.

Conclusion

In this study, we have employed a popular pre-trained deep learning CNN architecture (VGG16) to classify normal and abnormal brain MR images. We have obtained better performance than the rest of the techniques obtained using the same dataset. Our developed model can be used to find other brain abnormalities like Alzheimer's disease, stroke, Parkinson's disease, and autism.

Part 2: Brain Tumor Detection through Image Processing

Images classified as having tumors were considered for this part of the problem. Image processing was carried out using the OpenCV library in python 3.

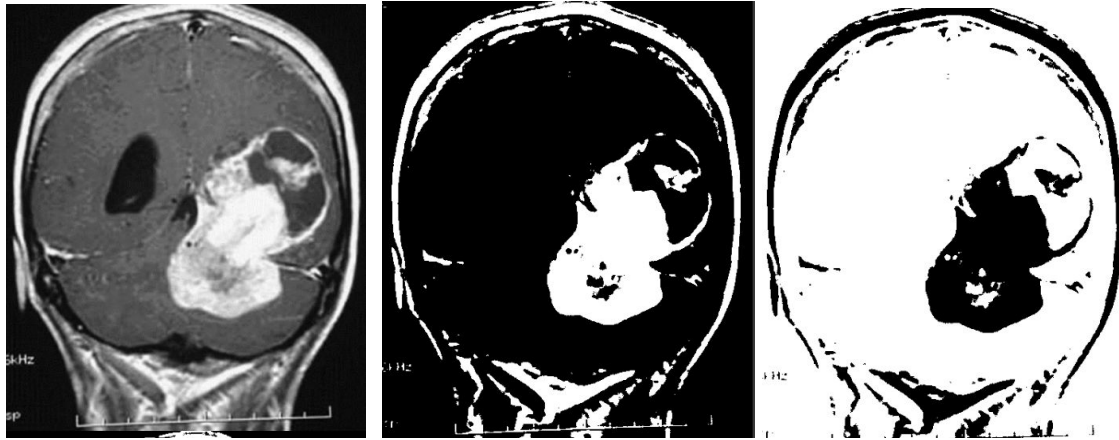


The image was converted to Grayscale and the obtained image was used for thresholding. Thresholding is a technique in OpenCV, which is the assignment of pixel values in relation to the threshold value provided. In thresholding, each pixel value is compared with the threshold value. If the pixel value is smaller than the threshold, it is set to 0, otherwise, it is set to a maximum value.

```
(T, thresh) = cv2.threshold(gray, 155, 255, cv2.THRESH_BINARY)
cv2.imshow('thresh', thresh)
```

Here we have used `cv2.THRESH_BINARY` in which if intensity is greater than the set threshold, value set to 255, else set to 0. `cv2.THRESH_BINARY_INV` is the opposite of `cv2.THRESH_BINARY`.

Here 155 is the threshold value and 255 is the maximum value that can be assigned.



(1) Grayscale Image

(2) Thresholding

(3) InvThresholding

The next step was applying Morphological operations to remove the unwanted part from the thresholding image.

```
kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (10, 5))  
closed = cv2.morphologyEx(thresh, cv2.MORPH_CLOSE, kernel)
```

Morphological transformations are simple operations based on the image shape. They are normally performed on binary images. They need two inputs, one is the original image and the second one is called a structuring element or kernel which decides the nature of operation.

The morphological operators used are Erosion and Dilation.

```
closed = cv2.erode(closed, None, iterations = 14)  
closed = cv2.dilate(closed, None, iterations = 13)
```

Erosion erodes away the boundaries of foreground objects. A pixel in the original image (either 1 or 0) will be considered 1 only if all the pixels under the kernel is 1, otherwise it is eroded. Therefore, the pixels near the boundary will be discarded depending upon the size of the kernel. So the thickness or size of the foreground object decreases or simply white region decreases in the image. Thus we have used this to remove the small white noises.

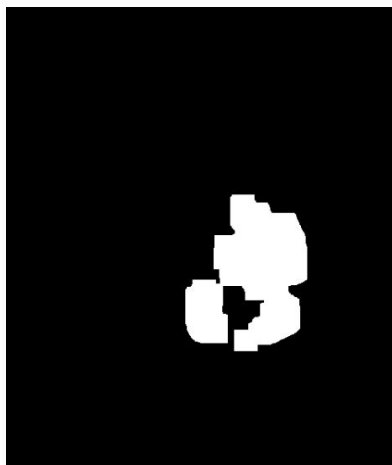
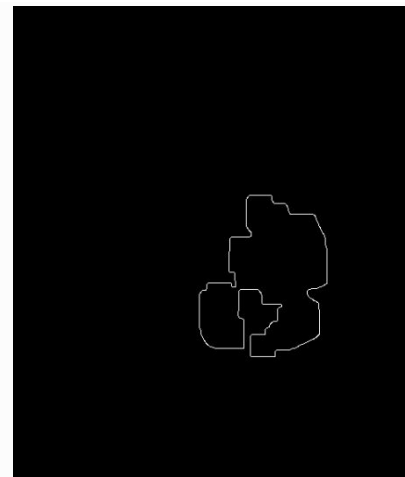


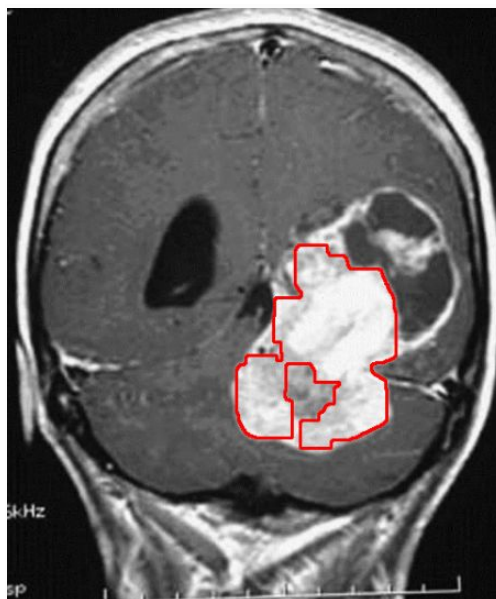
Image after erosion and dilation



Auto canny

```
edged = cv2.Canny(image, lower, upper)
canny = auto_canny(closed)
```

Canny edge detection is carried out on the image to find the outline of the tumor. Auto_canny function is defined to automatically calculate the lower and upper threshold values.



The final Image

The contour of the tumor is found and superimposed on the original Image.

```
(cnts, _) = cv2.findContours(canny.copy(), cv2.RETR_EXTERNAL,  
cv2.CHAIN_APPROX_SIMPLE)  
cv2.drawContours(image, cnts, -1, (0, 0, 255), 2)
```

Contours are defined as the line joining all the points along the boundary of an image that are having the same intensity. Contours come handy in shape analysis, finding the size of the object of interest, and object detection.

The `cv2.findContours` function takes an input of the source image and the contour approximation method and outputs the contours, and hierarchy. 'contours' is a Python list of all the contours in the image. Each individual contour is a Numpy array of (x, y) coordinates of boundary points of the object. `cv2.CHAIN_APPROX_SIMPLE` removes all redundant points and compresses the contour, thereby saving memory.

Thus we have successfully detected the tumor from the Brain MRI scan.