MRI-based Brain Tumor Detection using Covolutional Neural Networks

CSE 465, section 6
Group 7

1 Introduction

Classic Deep Neural Networks (DNNs), due to their relatively straightforward architecture, are unable to capture the contextualized information of images. The complexity of the DNNs grow exponentially when high resolution images are used as inputs. As an alternative, Convolutional Neural Networks (CNNs) are used, which perform significantly better for learning complex patterns in images. [1]. Therefore, we conduct this study for the purpose of detecting brain tumors from MRI scans using 12 pre-trained CNN models and 1 CNN model designed by us. The considered dataset [2] contains 4 classes of images (3 tumor types and 1 tumor-free) , namely, **glioma**, **meningioma**, **pituitary** and **no tumor**. The efficacy of each model in tumor detection has been compared based on their performance and training complexity, with which we finally determine an optimum model, among those considered, suited for this specific type of image detection.

2 Models and Training

Data Pre-processing

Prior to training the models, we designed a pre-processing pipeline for the images. However, the existing models support different input image size and color systems. Hence, keeping augmentation methods consistent across all models, we implemented pre-processing methods native to each model. For our custom CNN, we chose the input image as $299 \times 299 \times 1$, which is in the grayscale.

Some Key features of our pre-processing pipeline:

- **Data Augmentation**: The augmentation techniques used include transformations such as rotation, shifting, shearing, zooming, brightness variation, and flipping - which improve the model's ability to generalize and handle real-world variations of MRI scans.
- **Pre-processing with model Compatible Input**: Each pre-trained CNN expects input images in a format that matches the pre-processing used during its original ImageNet training. This pre-processing is model-specific. So, for that we used the appropriate pre-processing function that matches the model architecture being used, such as

$$from\ \ tensorflow.keras.applications.efficientnet\_v2\ \ import\ \ preprocess\_input \tag{1}$$

$$ImageDataGenerator(preprocessing\_function = preprocess\_input) \tag{2}$$

  This segment of code applies the preprocessing required by EfficientNetV2 (scaling, normalization) that ensures images are correctly prepared for feature extraction by the pre-trained model.
- **No augmentation for validation and test sets**: Validation and test sets use only the model specific pre-processing function without any transformation for evaluation-consistency and for avoiding randomness during validation or testing.

Table 1: Model hyperparameters that vary across models

Model	Input Shape	Dropout Rate	Pooling	Fully Connected Layers
DenseNet121	(299, 299, 3)	0.5 (dense layers)	GlobalAverage + GlobalMax	2 (1024, 128)
EfficientNetB0	(300, 300, 3)	0.4 (dense block)	GlobalAverage	2 (128, 64)
EfficientNetB3	(300, 300, 3)	0.3 (dense block)	GlobalAverage	2 (128, 64)
EfficientNetV2B1	(299, 299, 3)	0.3 (dense block)	GlobalAverage	2 (128, 64)
EfficientNetV2B3	(300, 300, 3)	0.3 (dense block)	GlobalAverage	2 (128, 64)
InceptionV3	(299, 299, 3)	0.5 (dense block)	GlobalAverage + GlobalMax	2 (1024, 128)
ResNet50V2	(300, 300, 3)	0.3 (dense block)	GlobalAverage	2 (1024, 128)
ResNet101	(300, 300, 3)	0.5 (dense block)	GlobalAverage	3 (512, 512, 128)
VGG16	(299, 299, 3)	0.5 (dense block)	GlobalAverage	2 (512, 128)
VGG19	(299, 299, 3)	0.3 (dense block)	GlobalAverage	2 (512, 128)
Xception	(299, 299, 3)	0.3 (dense block)	GlobalAverage	2 (128, 128)
AlexNet	(227, 227, 3)	0.5 (dense layers)	MaxPooling2D	2 (2048, 2048)
Custom CNN	(299, 299, 1)	0.25, 0.3, 0.4, 0.5	MaxPooling2D	2 (256, 4)

Initialization of the model

The model initialization for each pre-trained model is somewhat different apart from EfficientNetV2B1 and EfficientNetV2B3. The building part of these two models is kept the same intentionally as EfficientNetV2B3 is just a more powerful and larger model. Before training, some of model hyperparameters are varied to better optimize the respective models (table 1), while the following hyperparameters are kept same for the models

- **Learning Rate Schedule**: An ExponentialDecay learning rate schedule is used to gradually reduce the learning rate during training, allowing faster learning at the start and finer adjustments later to improve convergence.
- **Freezing and Unfreezing Layers**: Some early layers or sometimes initial layers were frozen and then gradually unfrozen for helping the model better adapt to the brain tumor classification task.
- **Adam Optimizer**: Models use the Adam optimizer with a scheduled learning rate. Adam dynamically adjusts learning rates for individual parameters, improving training efficiency and stability.
- **ModelCheckpoint**: Saves the model with the best validation loss during training, ensuring that the best-performing version is preserved.
- **EarlyStopping**: Monitors validation loss and stops training if no improvement is seen for 10 epochs, helping to prevent overfitting and saving training time.

Following model initialization, we move onto training the models with the preprocessed images. We have set the maximum number of epochs to 1000 for all models, however early stopping was used in each case. EarlyStopping is a built-in callback in TensorFlow that automatically stops training when a monitored performance metric (like validation loss or accuracy) stops improving. Hence, the 1000 epoch ceiling was set to provide the models with appropriate leeway to reach the true minimum validation loss.

Custom CNN architecture

The model follows a deep CNN structure, incorporating multiple convolutional blocks, each followed by batch normalization, pooling, and dropout layers to enhance learning and prevent overfitting. The architecture progressively increases the depth and complexity of learned features.

- **Input Layer**
  - Shape: (299, 299, 1)
  - Grayscale MRI input images
- **Convolutional Block 1**
  - Two Conv2D ayers with 32 filters, kernel size of (3x3), ReLU activation
  - BatchNormalization after each convolution
  - MaxPooling2D with a (2x2) window
  - Dropout with a rate of 0.25
- **Convolutional Block 2**
  - Two Conv2D layers with 64 filters, kernel size of (3x3), ReLU activation
  - BatchNormalization after each convolution
  - MaxPooling2D with a (2x2) window
  - Dropout with a rate of 0.3
- **Convolutional Block 3**
  - Two Conv2D layers with 128 filters, kernel size of (3x3), ReLU activation
  - BatchNormalization after each convolution
  - MaxPooling2D with a (2x2) window
  - Dropout with a rate of 0.4
- **Global Feature Aggregation**
  - GlobalAveragePooling2D to reduce spatial dimensions and capture global features
- **Fully Connected Layer**
  - Dense layer with 256 units and ReLU activation
  - BatchNormalization
  - Dropout with a rate of 0.5
- **Output Layer**
  - Dense layer with num_classes = 4, using softmax activation for multi-class classification
- The model uses the Adam optimizer with an exponentially decaying learning rate.

3 Results and Analysis

The pipeline of our study includes testing after the models are trained. The trained models were deployed on a test set of MRI scans and we recorded the accuracy, precision and recall of all the models. Furthermore, to maintain statistical significance, we ran the whole training process three times for each model, every time recording the performance metrics mentioned (the excel sheet containing the metrics of the 3 runs is provided as supplement). From these, we provide the mean metrics with their standard deviations in table 2. Among all the models ran, it was observed that EfficientNetV2B1 demonstrated the best accuracy with the least fluctuating accuracies and losses (fig. 1a, fig. 2a) and the least number of parameters from the best performing models.

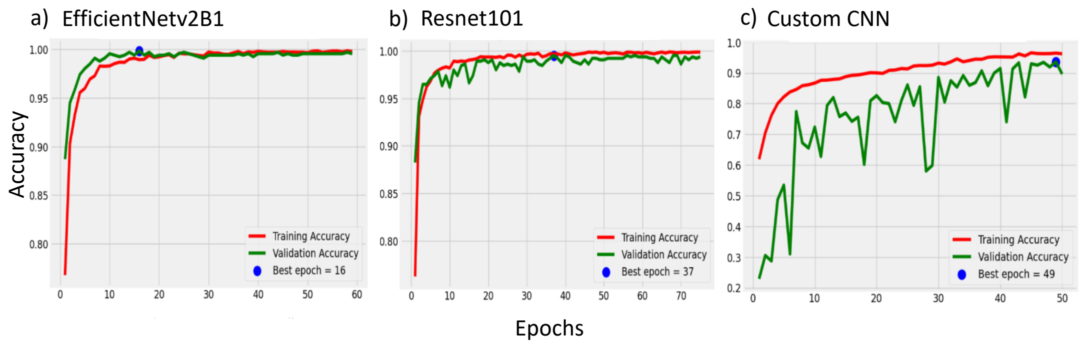

Figure 1: Training and Validation accuracy for a) EfficientNetV2B1, b) ResNet101, c) Custom CNN

ResNet101, on the other hand, has the most number of parameters but it performs almost close to but not better than EfficientNetV2B1 (table 2). Therefore, EfficentNetV2B1 is the best performing pre-trained model with being more efficient and interpretable than the most parameter dependent model. Moreover, both models require almost the same time (about 30 minutes) to train.

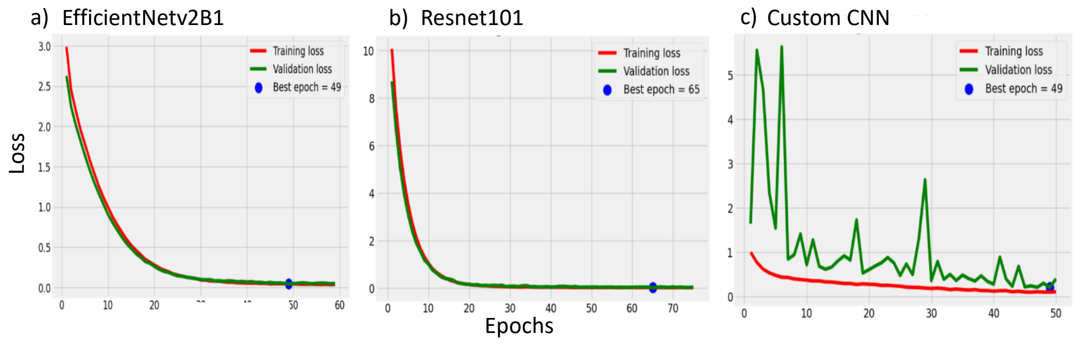

Figure 2: Training and Validation Loss for a) EfficientNetV2B1, b) ResNet101, c) Custom CNN

Now, considering the custom CNN, which has a significantly lesser number of parameters than EfficientNetV2B1, it still gives a high degree of accuracy. There are visibly great fluctuations in the loss and accuracy in the early epochs of the custom CNN (fig. 1c, fig. 2c) compared to both the EfficientNetV2B1 and ResNet101. However, with the progression of training, the model performance improves to finally achieve an average accuracy of 98.22%. Owing to the simplicity of the model compared all the others, it is much more efficient in training than all the other models considered.

Table 2: Comparison of the performance of various models

Models	Accuracy %	Precision %	Recall %	Parameter list (M)
DenseNet121	99.647 ± 0.192	99.647 ± 0.192	99.647 ± 0.192	8.748
EfficientNetB0	99.593 ± 0.075	99.593 ± 0.075	99.593 ± 0.075	4.228
EfficientNetB3	99.950 ± 0.071	99.950 ± 0.071	99.950 ± 0.071	11.036
EfficientNetV2B1	99.950 ± 0.071	99.950 ± 0.071	99.950 ± 0.071	7.101
EfficientNetV2B3	99.900 ± 0.141	99.900 ± 0.141	99.900 ± 0.141	13.134
InceptionV3	99.746 ± 0.191	99.746 ± 0.191	99.746 ± 0.191	26.134
ResNet50V2	99.593 ± 0.075	99.690 ± 0.122	99.593 ± 0.075	27.896
ResNet101	99.746 ± 0.191	99.746 ± 0.191	99.746 ± 0.191	45.610
VGG16	99.750 ± 0.071	99.750 ± 0.071	99.750 ± 0.071	16.423
VGG19	98.527 ± 0.519	98.577 ± 0.473	98.423 ± 0.577	20.090
Xception	99.747 ± 0.191	99.747 ± 0.191	99.747 ± 0.191	21.124
AlexNet	99.796 ± 0.192	99.796 ± 0.192	99.796 ± 0.192	7.101
Custom CNN	98.223 ± 0.258	98.070 ± 0.316	98.173 ± 0.249	0.323

4 Explainable AI: LIME

We used LIME to explain model's prediction. LIME stands for Local Interpretable Model-agnostic Explanations. It is a powerful technique used to understand and explain predictions made by complex machine learning models—especially black box models like deep neural networks.

Fig. 3 shows that LIME has outlined regions in yellow — these are the superpixels or image segments that most influenced the model's prediction of "meningioma". The LIME-highlighted regions form a ring-like contour around the outer brain. This is very consistent with where a meningioma would appear. From table 3, it can interpreted that after applying LIME on the trained custom CNN, the model extracts the regions of the image that are most important to detecting the tumors. The yellow marked regions show in fig. 3-right shows this. Hence, the model is interpretable and is extracting the correct features.

Table 3: Summary of how LIME explains the relevant regions of the image

Region	**What is in MRI**	**What LIME shows**	**Clinical significance**
Outer edges of the brain	Displacement of brain tissue	Yellow LIME contours	Likely tumor location (extra-axial)
Central brain structures	Symmetrical (relatively)	No LIME highlight	Not relevant for extra-axial tumor

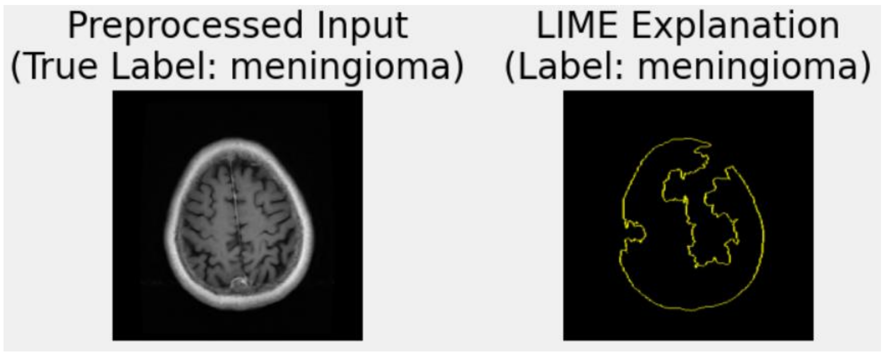

Figure 3: The outcome of applying LIME on the trained custom CNN

5 Conclusion

In conclusion to this study, most of the models that we trained to detect brain tumors from MRI images achieved a considerable degree of accuracy in prediction. Interestingly, the custom CNN is also up to par with the pre-trained models. Therefore, with the custom CNN, we achieve a trade-off between accuracy and simplicity of the model. Due to computational resource constraint, we had to limit our experiments, however, it is a viable to avenue to explore further in order to improve model performance while keeping it simple.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[2] M. Nickparvar. Brain tumor mri dataset, 2021.