Investigating the Effects of Transfer Learning on Multi-class Brain Tumor Classification

Ontario Tech University
CSCI 5770 - Machine Learning Faculty of Science

Abhinav Sharma

1 Introduction

Brain tumor diagnosis and classification has been a crucial topic of study in medical image analysis because the outcomes can directly impact patient care and treatment options. However, accurately diagnosing and classifying brain tumors can be a challenging task due to the complexity and variability of the brain tumor images. In recent years, deep learning approaches have shown promising results in medical image analysis, especially in image classification tasks. Deep neural networks have the ability to learn and extract relevant features from images, which can lead to improved accuracy and efficiency in image analysis. Transfer learning is a popular technique in deep learning that allows leveraging the pre-trained knowledge of a neural network on a large dataset to improve the performance of the model on a target dataset. By using a pre-trained model as a starting point, transfer learning can speed up training time, reduce the need for large amounts of labeled data, and improve model generalization. More emerging research is being done these days on Local Interpretable Model-Agnostic Explanations (LIME) models as well. With the help of LIME, we can get some insights into how these classifiers are making decisions when processing images for classification purposes. Using LIME we can understand which regions of the MRI image does the classifier detect as a possible area of presence of the 'object', in this case a type of tumor. In this project, I aim to investigate the effect of transfer learning on brain tumor classifiers. By leveraging pre-trained models on a large dataset such as ImageNet, I can evaluate the effectiveness of transfer learning in improving the accuracy of different brain tumor classification models (various Architecture). I will compare the models without transfer learning with the pre-trained models (with Transfer learning). I will also compare the types of transfer learning with each other, types of fine-tuning a pre-trained model. LIME models will also be explored in an attempt to see where and how the learning is happening. The investigation can provide valuable insights into the use of deep learning techniques for medical image analysis, which can potentially improve patient outcomes and healthcare services. The .pynb scripts for all the models and the results can be seen from my Github repository. [7]. I got some ideas from the following references regarding certain code implementation procedures [10] [11].

2 Related Work

The following are some of the many related works that has been done on brain tumor classification with transfer learning and LIME explanations. A Transfer Learning approach for AI-based classification of brain tumors Mehrotra et al. propose an AI-based classification of brain tumors using Deep Learning algorithms on MRI data. The study utilizes publicly available datasets that classify brain tumors as malignant or benign. The proposed algorithm achieves an accuracy of 99.04% in classifying brain tumors, demonstrating its potential for accurately categorizing brain tumors. The study highlights the relevance of AI in medical imaging and the remarkable performance of DL algorithms in segmenting and classifying brain tumors. Brain tumor classification using deep CNN features via transfer learning The approach here by Deepak et al. [2] is using a pre-trained GoogLeNet to extract features from MRI images and integrates proven classifier models to classify the features. The system achieves a mean classification accuracy of 98%, outperforming state-of-the-art methods. The study also evaluates the system's performance with feIr training samples and finds that transfer learning is useful when medical image availability is limited. The paper discusses misclassifications and other performance measures such as AUC, precision, recall, F-score, and specificity. Overall, the study demonstrates the effectiveness of deep transfer learning for brain tumor classification in medical applications. Multi-grade brain tumor classification using deep CNN with extensive data augmentation Sajjad et.al [3] explore novel convolutional neural network based multi-grade brain tumor classification system to assist radiologists in the analysis of MRI. The

proposed system employs a deep learning technique to segment tumor regions from an MR image and extensive data augmentation to address the lack of data problem in multi-grade brain tumor classification. A pre-trained CNN model is fine-tuned using augmented data for brain tumor grade classification. The system is experimentally evaluated on both augmented and original data, and results demonstrate its convincing performance compared to existing methods. The study highlights the effectiveness of deep learning techniques and data augmentation for multi-grade brain tumor classification in medical imaging. Explanation-Driven Deep Learning Model for Prediction of Brain Tumor Status Using MRI Image Data Gaur et al[4] propose an explanation-driven deep learning model for predicting discrete subtypes of brain tumors (meningioma, glioma, and pituitary) using MRI images. The model utilizes a convolutional neural network (CNN), local interpretable model-agnostic explanation (LIME), and Shapley additive explanation (SHAP) for interpretation and accuracy. The dual-input CNN approach overcomes the classification challenge with images of inferior quality by adding Gaussian noise. The CNN training results show 94.64% accuracy compared to state-of-the-art methods. SHAP ensures consistency and local accuracy for interpretation, while LIME illustrates how the model operates in the immediate area. The study's emphasis is on interpretability and high accuracy, critical for developing trust and integration into clinical practice. The proposed method has vast clinical application potential for mass screening in resource-constrained countries. The paper highlights the importance of explanation and interpretability in DL models for medical imaging applications.

Various studies have been conducted in this domain, demonstrating the potential of deep learning algorithms and transfer learning for accurate brain tumor classification with MRI data. The studies use pre-trained models to extract features from MRI images and integrate proven classifiers to classify the features. The studies emphasize the importance of interpretation and accuracy in DL models for medical imaging applications. They suggest that transfer learning can be used to improve the accuracy of brain tumor classification models, and that interpretation techniques such as LIME can provide insights into the behavior of the model and its decisions.

3 Methodology

3.1 Proposed Method

I will use three pre-trained models and do two types of fine-tuning (explained in 3.4) on the target dataset (transfer learning) and compare them to the models without transfer learning (same architecture but randomly initialized weights). The three model architectures I chose from several publicly available models are: Vgg16, ResNet50 and InceptionV3. For all the training the hyper-parameters are kept fixed for the sake of comparison. I understand that they can be tuned or changed for better performance. Then we will use LIME to get a heat map for a sample image for each classifier and compare them with each other to try and see what the classifier learned.

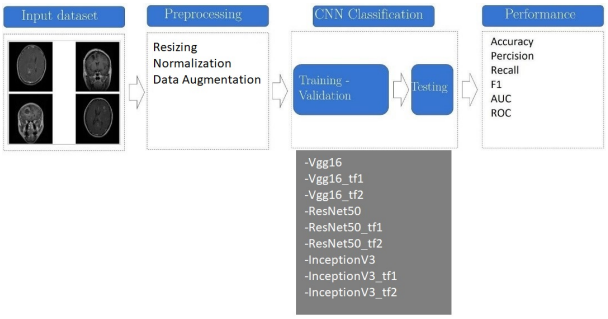

Figure 1: Flow Chart of the Methodology

3.2 Preprocessing

Python version 3.9.16 was used in the Google Colab platform to do the preprocessing, model building and testing. A standard GPU which is available on Colab: NVIDIA T4 Tensor Core GPU was used.

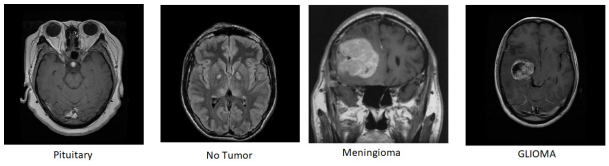

Figure 2: samples of the 4 classes from the Target data set

The dataset is from kaggle [6] , containing 7023 images with four classes: glioma - meningioma - no tumor and pituitary.

The preprocessing steps taken for the Brain Tumor MRI dataset include resizing the images, reading and labeling the images, converting the labels to numerical values, and splitting the dataset into training and testing sets. Image cropping was attempted Resizing the images is an important preprocessing step to ensure that all images have the same dimensions, which is necessary for inputting the images into a neural network. In this code, the function resize_image uses the cv2.resize function from the OpenCV library to resize each image to a fixed size of (128 x 128) pixels using the interpolation method cv2.INTER_AREA. All the models can take a custom input tuple of (128,128) since the include_top parameter is set to False. More about this in section 2.3. Reading and labeling the images is another important preprocessing step. There are many ways to do it, I chose to do stick to the following method where looping through the folders and sub-folders of the dataset path and adding each image to a list of images along with its corresponding label. In the code, the function read_dataset checks if the folder is either 'Testing' or 'Training', and then checks if the sub-folder is one of the four categories: 'glioma', 'meningioma', 'notumor', 'pituitary'. For each image file that ends with '.jpg', it creates the path to the image, resizes the image using the resize_image function, and adds it to the list of images. It also adds the corresponding label to the list of labels. Converting the labels to numerical values is necessary for classification tasks, as neural networks require numerical values as inputs. In this code, the function text_to_index maps the text labels to numerical values by creating a list of index labels based on the target list. Next, splitting the dataset into training and testing sets is important for evaluating the performance of the neural network on unseen data. In this code, the train_test_split function from the sklearn.model_selection module is used to split the images and labels into training and testing sets with a test size of 0.2 and stratified sampling based on the labels.

Finally, The code snippet is using the Keras library to create an instance of ImageDataGenerator with rotation and flip parameters. This is a preprocessing step that applies data augmentation to the training images. Data augmentation is a technique used to artificially increase the size of the training dataset by creating new variations of existing images. In this case, rotation_range=15 means that the images will be randomly rotated between -15 to 15 degrees. horizontal_flip=True and vertical_flip=True indicate that the images may be flipped horizontally or vertically, respectively. By using these techniques, the model can learn to generalize better to unseen images and reduce overfitting. The batch size is also set to 32, which means that the training data will be fed to the model in batches of 32 images at a time. This helps to speed up the training process and improves the efficiency of memory usage. There are other data augmentation parameters that can be initialized but I choose to stick with the three that are commonly chosen in training with a medium data set size.

Overall, the preprocessing steps in this code ensure that the images are standardized in terms of size and labeling, and the dataset is properly split into training and testing sets. These preprocessed data are then used to train and evaluate a neural network for classification.

3.3 Training Without Transfer Learning

Vgg16, ResNet50 and InceptionV3 are the three model that will be trained without loading the pretrained weights into any of these model architectures. When training the models they will be named model_"non_tf". ResNet-50 is a popular choice for transfer learning due to its excellent performance in the ImageNet-based ILSVRC 2015 ranking challenge. It has 50 layers, which are arranged in a specific way to allow for the use of residual connections. The first layers of the network perform simple operations like convolution and pooling, while the later layers learn more complex features. Its residual connections enable the network to learn more complex and subtle features useful for image classification tasks, and prevent the gradients from becoming too small. It is effective in learning from large datasets and has a proven residual learning block architecture shown in Figure 4.

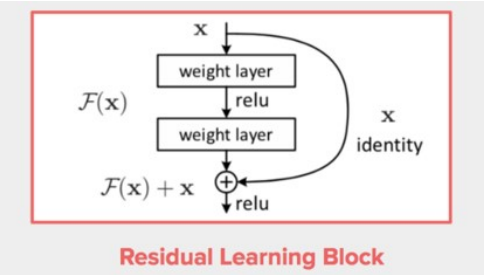

Figure 3: A Residual Learning Block

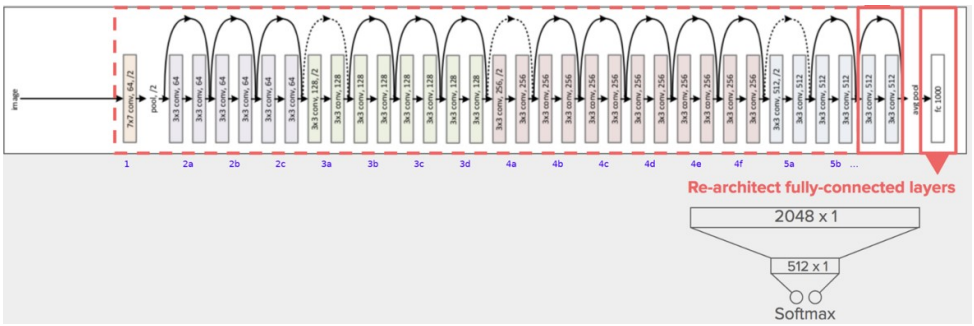

Figure 4: Full Architecture of ResNet50

When training the model, the architecture of the three keras pre-trained models are used, initializing the weights randomly by setting the 'weights' parameter to 'None'. Then the all the layers (the convolutional base of each mode and the subsequent functional layers/fully connected layers) are trained on the target dataset. Figure 5 shows a snippet of the python implementation of this of the base of the Resnet50. The fully connected layers are omitted as we want to add our own functional layers/ fully connected layers, can be seen from the code [7].

```
base_model = ResNet50(weights=None, include_top=False, input_shape=(128, 128, 3))

# Set ResNet50 layers to be non-trainable
for layer in base_model.layers:
    layer.trainable = True
```

Figure 5: Loading ResNet50 convolutional base architecture with randomly initialized weights and no fully connected layers

The same method of training is applied to InceptionV3 model from Google whose architecture is shown in Figure 6. The convolutional base with the randomly initialized weights are trained along with custom added functional layers. The convolutional based used in the training is the highlighted with the red border in the below figure.

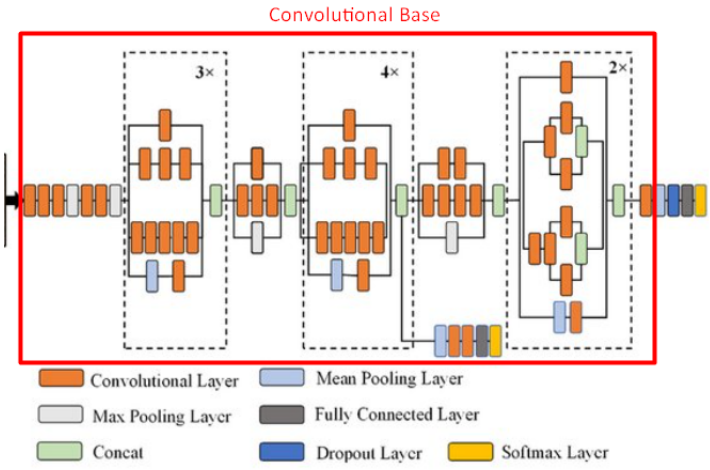

Figure 6: InceptionV3 Architecture

InceptionV3 has 48 layers and is more complex than ResNet50 due to the use of factorization of convolutions and convolutional filters of different sizes. It enables the learning of complex features at multiple scales, has fewer parameters to prevent overfitting, and uses batch normalization to improve generalization performance. While ResNet50 uses residual connections to help with vanishing gradient issues, InceptionV3 has a more complex architecture. However, this can lead to longer training times and higher computational requirements.

Lastly, the VGG-16 is also trained the same way. The network architecture can be seen in Figure 7 below.

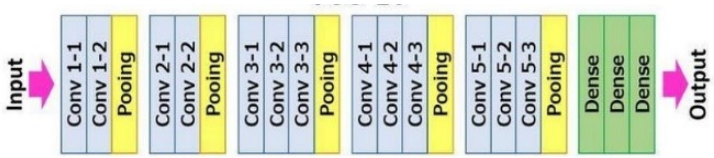

Figure 7: VGG-16 Architecture

The VGG-16 network has 13 convolutional layers and 3 fully connected layers, with a 3x3 filter size for convolution. It is a simpler architecture compared to ResNet50 and InceptionV3 as all layers use the same convolutional filter size and pooling size. Despite having fewer layers, VGG-16 has many parameters due to smaller filters. It is highly accurate on image classification tasks, especially when trained on large datasets. One advantage of VGG-16 is its simplicity and ease of implementation, but it may not perform as well on more complex tasks that require deeper and more complex networks.

3.4 Training with Transfer Learning

There are two types of fine-tuning that will be investigated, in this report I will call them Type 1 and Type 2. Type 1 would be where the layers of convolutional base of the pre-trained models are frozen and just the fully connected output layer is trained on the target dataset. Type 2 Fine-Tuning will be when the initial weights of the pre-trained model are retained. We retrain the entire model (all the layers) on the brain tumor images using the weights initialized as in the pre-trained model. I will apply both type 1 and type 2 fine-tuning in all the models. During training both these types will be referred to as model_tf1 and model_tf2. Both types of fine-tuning can be demonstrated by Figure 8. For Type 1, layers from Layer 1 to (L-1) are frozen with ImageNet weights and only the Output layer is trained from scratch on the target dataset. For Type 2 however, all the layers are trained from Layer 1 to Output layer after being initialized with the ImageNet weights. So the imageNet weights are being re-trained in Type 2.

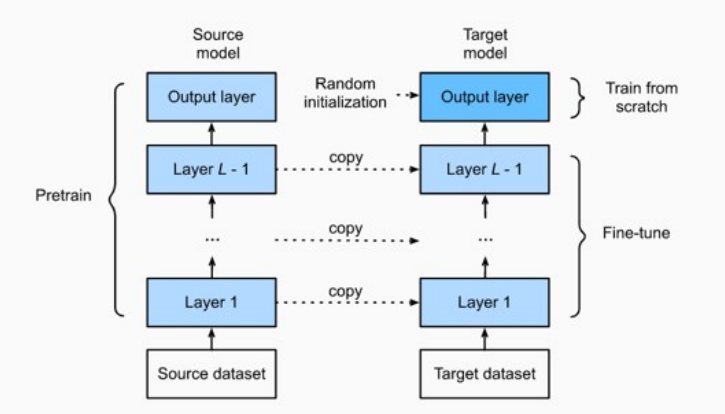

Figure 8: Fine- Tuning output layer

During training, the input image tuple is (128,128) and the include_top = False. What this does is that it lets you select if dense layers are needed, 'whether to include the 3 fully connected layers at the top of the network' [4]; the three fully connected layers in this case would be the 'Output layer' on the left side in Figure 8. So we set it to False because it is not needed in this case. Instead, 3 functional layers are added to the convolutional base. This is illustrated from the code in Figure 9.

```
base_model = ResNet50(weights='imagenet', include_top=False)
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu')(x)
predictions = Dense(4, activation='softmax')(x)
model = Model(inputs=base_model.input, outputs=predictions)
```

Figure 9: Setting up and Initializing the Model Parameters for ResNet50

• The base_model = ResNet50(weights='imagenet', include_top=False) line loads the ResNet50 model with pre-trained weights from the ImageNet dataset, and sets include_top=False to exclude the final classification layer of the original network.

• The x = base_model.output line retrieves the output tensor from the last convolutional layer in the base model.

• The x = GlobalAveragePooling2D()(x) layer performs global average pooling on the output tensor, which reduces the spatial dimensions of the tensor to 1x1 and collapses the channel dimensions. This results in a 2D feature map that summarizes the presence of features throughout the entire image.

• The x = Dense(1024, activation='relu')(x) layer applies a fully connected neural network layer with 1024 neurons and ReLU activation function to the output of the global average pooling

• Finally the output layer, predictions = Dense(4, activation='softmax')(x) layer applies another fully connected neural network layer with 4 output neurons (equal to the number of classes in the dataset) and a softmax activation function to obtain the final output probabilities for each class. After this we either do Type 1 or Type 2 training by switch the 'layer.trainable' parameter of the base model to 'True' and 'False' as shown in the code below.

```
for layer in base_model.layers:
    layer.trainable = False;
```

The Type 1 and Type 2 transfer learning method is applied to all three models and the results would be compared with the non-transfer learning, with training the model from scratch approach.

3.5 Performance Metrics

These are the outcomes from a performance matrix: True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN) are commonly used terms in performance metrics. In brain tumor classification, precision and recall are particularly important metrics. False positives and false negatives can have serious consequences, and high precision and recall are necessary to reduce these errors. F1 score is also a good metric to use as it gives equal weightage to both precision and recall. ROC-AUC can also be useful in assessing the overall performance of the classifier.

3.6 LIME Implementation

Local Interpretable Model-Agnostic Explanations or LIME is a way of understanding how models work. For this project the LIME Python library is used. Lime helps to interpret the behavior of machine learning models by providing local and interpretable explanations of their decisions. The "Local" refers to the fact that LIME focuses on interpreting the behavior of one data point at a time, not every data point in the dataset. In image classification tasks, LIME interprets one image at a time. The "Interpretable" means that LIME provides explanations of what causes the model to make its decisions. The "Model-agnostic" means that LIME works on any kind of black box classifier. LIME, a process for interpreting machine learning models, is composed of four main steps:

• Local Instance Picking and Artificial Data Generation: LIME randomly selects a data point or image and generates a series of artificial data points. In image classification tasks, the image is divided into superpixels, groups of pixels with similarities. Some superpixels are randomly turned off or made to appear black to create multiple artificial images, a process known as image perturbation.

• Artificial Data Point Prediction: The machine learning model initially used to classify the images predicts a class for each perturbed image and each superpixel.

• Calculation of Distance and Weight Assignment: The cosine similarity is used to calculate the distance between each perturbed image and the random image being explored. A high value of cosine similarity indicates a smaller angle between the two data points, and thus greater similarity.

• Use of a Linear Classifier to Classify the Perturbed Images: After calculating the weights for each perturbed image, a linear classifier is fit to the perturbed images, the labels for each superpixel of those perturbed images, and the weights. The linear model then ranks the coefficients of each superpixel to determine which superpixels are most influential for classifying the image.

In the case of brain tumor classification, LIME can help to shed light on the features that the model is using to classify the images. Brain tumors can have various appearances in MRI scans, which can make classification difficult. By using LIME on a convolutional neural network trained on brain MRIs, the most important superpixels and features used by the model can be identified and analyzed. I got this idea of using LIME on the output of my models and visually inspect the LIME heat maps as shown in this article [8].

4 Experiments and Results

4.1 Pre-Result Discussion

There are many ways to fine-tune a pre-trained model, like how we did in our experiment. One way can be where the output layer is removed, and the rest of the network is used as a fixed feature extractor for the new dataset. Another method is to use the architecture of the pre-trained model but initialize all the weights randomly and train the model according to the new dataset. Another one is to freeze the initial layers of the pre-trained model and retrain only the higher layers that are customized for the new dataset. Three ways described above.

There are different cases where these techniques can be applied [8]:

• Case 1: When the size of the dataset is small and the data similarity is very high, customize and modify only the output layers according to the problem statement. Use the pre-trained model as a feature extractor.

• Case 2: When the size of the dataset is small and the data similarity is very low, freeze the initial layers of the pre-trained model and train only the higher layers customized for the new dataset.

• Case 3: When the size of the dataset is large, but the data similarity is very low, train a new neural network from scratch using the new dataset.

• Case 4: When the size of the dataset is large and there is high data similarity, retain the architecture and initial weights of the pre-trained model, and retrain it using the weights as initialized in the pre-trained model.

So with this theory in mind and relating it to this project,where the size of the target dataset is small and the similarity between the target dataset and source dataset is low, the best performing model should be 'tf1', which freezes the convolutional base of the pre-trained model with ImageNet weights and only trains the added fully connected layers. This is because in Case 2, the recommendation is to freeze the initial layers of the pre-trained model and train only the remaining layers, which is exactly what 'tf1' does. This approach is effective when the target dataset has low similarity to the source dataset, as it allows the model to benefit from the pre-trained weights of the convolutional layers while still adapting to the new data through training the added fully connected layers. The second-best performing model would likely be 'tf_2', which re-trains all the layers including the added fully connected layers using the pre-trained model with ImageNet weights. However, since the target dataset has low similarity to ImageNet, there may be limited benefits from the pre-trained weights, and the model may overfit on the small target dataset. The least performing model would likely be 'non_tf', which uses the architecture of Model with randomly initialized weights and trains all the layers. Since there are no pre-trained weights in this case, the model may not perform as well as the other two models, especially on a small dataset.

4.2 Results and Interpretations

In the first part of this section we will compare the results of the three types of training done for each model architecture. In the second part we will compare the different architecture type with one another. Figure 10 below displays the results of all 9 models of test data.

Model	Loss	Accuracy	Precision	Kappa
ResNet-tf1	0.8592376708984375	68.33568215370178	68.33568406205924	57.34121094888505
ResNet-not-tf	0.047053027898073196	99.01269674301147	99.01269393511988	98.68081978956378
ResNet-TF2	0.07701630890369415	97.95486330986023	97.95486600846263	97.2682165949099
VGG16-tf1	0.4137129485607147	88.43441605567932	88.43441466854725	84.5468898682161
VGG16-not-tf	1.383479118347168	28.208744525909424	28.208744710860366	0.0
VGG16-tf2	0.03728744387626648	98.87164831161499	98.8716502115656	98.49246064204644
Inception-tf1	0.3969716727733612	88.99859189987183	88.99858956276445	85.30857843055864
Inception-not-tf	0.031023116782307625	98.94217252731323	98.94217207334273	98.58648461315069
Inception-tf2	0.023205656558275223	99.4358241558075	99.4358251057828	99.24623583037648

Figure 10: Results table displaying the performance metrics scores for all 9 models.

4.2.1 Comparing the training types within and across the model architectures

ResNet50 Looking at the table of results for the ResNet models, it is surprising to see that the 'ResNet-tf1' model performed the worst in terms of both accuracy and precision, while the 'ResNet-not-tf' model performed the best. One possible reason for this could be that the target dataset used in this project is significantly different from the ImageNet dataset on which the ResNet model was pre-trained. This means that the pre-trained weights of the convolutional layers may not be as beneficial for the target dataset as they would be for a dataset that is more similar to ImageNet. In this case, training all the layers of the model (as in 'ResNet-not-tf') from scratch using the target dataset may result in a better model than only training the added fully connected layers (as in 'ResNet-tf1') or retraining all the layers with the pre-trained weights (as in 'ResNet-TF2').

Another possible reason could be related to the size of the target dataset. Since the size of the target dataset is small, it may not be sufficient to fine-tune the pre-trained weights of the ResNet50 model. In this case, training all the layers from scratch (as in 'ResNet-not-tf') may be more effective in capturing the specific patterns and features of the target dataset than only training the added fully connected layers (as in 'ResNet-tf1') or retraining all the layers with the pre-trained weights (as in 'ResNet-TF2'). It is also possible that the architecture of the ResNet50 model may not be the

best fit for the target dataset. In this case, training all the layers from scratch (as in 'ResNet-not-tf') may allow for more flexibility in adapting the architecture to the specific patterns and features of the target dataset. VGG16 VGG16 models have different performance compared to the ResNet models, which could be due to architectural differences between the two models. The best performing model for VGG16 is actually 'tf2', which contradicts our hypothesis from 4.1. One reason for this could be that VGG16 has a simpler architecture compared to ResNet, and thus may benefit more from retraining all the layers, including the convolutional base, using the pre-trained weights. The second-best performing model is 'tf1', which is consistent with our hypothesis. The target dataset still benefits from the pre-trained convolutional base, but only the added fully connected layers are trained to adapt to the new data. The worst performing model is 'non_tf'. Without any pre-trained weights, the model has to learn everything from scratch, which may not be optimal for a small dataset with low similarity to the source dataset trained on a small architecture. InceptionV3 the performance of the Inception models is generally higher compared to the ResNet50 and VGG16 models. This could be attributed to the fact that Inception architecture is designed to be computationally efficient and has been shown to perform well on image classification tasks. Similar to the ResNet50 and VGG16 models, the performance of the tf1 Inception model is lower compared to the tf2 model.

This is likely due to the fact that the tf1 model only trains the added fully connected layers while freezing the convolutional base, which may limit its ability to adapt to the new target dataset. In contrast, the tf2 model re-trains all the layers including the convolutional base, allowing it to better adapt to the new dataset. The non-tf Inception model performs just slightly worse than the non-tf ResNet50. This is just a very slight difference and we can't really point to a reason. Both models are large and unique in their own way and perform well when training the entire architecture from scratch approach, on the target dataset.

Overall, the results suggest that the Inception architecture may be better suited for small datasets with low similarity to the source dataset, as it is designed to be computationally efficient and has shown to perform well on image classification tasks. The performance of the tf2 Inception model suggests that re-training all the layers including the convolutional base may be beneficial in adapting the pre-trained model to the new target dataset. However, this may not always be the case and the optimal approach may depend on the specific characteristics of the target dataset

4.2.2 Parameters and Layers

Model	Total Parameters	Trainable Parameters	Non-Trainable Parameters
inception_not_tf	23905060	23905060	0
resnet_not_tf	25689988	25689988	0
vgg16_not_tf	15244100	15244100	0
inception_tf2	23905060	23905060	0
inception_tf1	23905060	2102276	21802784
vgg16_tf2	15244100	15244100	0
vgg16_tf1	15244100	529412	14714688
resnet_tf2	25689988	25689988	0
resnet_tf1	25689988	2102276	23587712

Figure 11: Model Parameters of all 9 Models

By looking at the parameters table, we can confirm that for non_tf and tf2 models, all the layers are being trained as there are no non-trainable parameters. There are many other factors of-course contributing to the performance of the models but for large model architectures (like resnet and inception) the non_tf and tf2 models do seem to perform better than the tf1 type. As expected, there's a correlation of some strength between the number of trainable parameters and the performance of the model.

Model Name	Conv Layers	Dense Layers	Activation Layers	Normalization Layers	Other Layers
inception_not_tf	94	2	94	94	30
resnet_not_tf	53	2	49	53	21
vgg16_not_tf	13	2	0	0	7
inception_tf2	94	2	94	94	30
inception_tf1	94	2	94	94	30
vgg16_tf2	13	2	0	0	7
vgg16_tf1	13	2	0	0	7
resnet_tf2	53	2	49	53	21
resnet_tf1	53	2	49	53	21

Figure 12: Number of layers across different architectures

For further visualization in the code, to see the exact model architecture layer by layer and their input / output breakdown by layer, the Hack-Regular.ttf font is being used to generate the layered view and

the plot of the Keras model. The font is specified using the ImageFont.truetype() function from the PIL library, and is passed to the visualkeras.layered_view() function along with the model to generate an image of the model's architecture. The keras.utils.plot_model() function is also used to generate a separate plot of the model. Both of these images are saved as PNG files and then opened using the Image() function from IPython.display to be displayed in the notebook. End of 'Results_MRI.ipynb' in my GitHub repo[githubrepo].

To be fair all the hyper-parameters for training across all models was kept fixed.From figure 12 we can see that VGG16 has considerably smaller with only 13 Convolutional layers. Hence when training from scratch, with the fixed parameters, its simple architecture was not enough to learn appropriate features hence the performance was terrible. To test this, I added multiple sequential layers and functional layers (convolutional, pooling and dense layers) a top my VGG16 base as seen in 'modification_VGG16_MORELAYERS.ipynb' in [7]. The results can be seen below in Figure 13 from the classification report. Results are significantly better when the sequential layers are added. These scores are significantly better than the ones I got earlier, 28% for accuracy and precision.

Classification Report:

	precision	recall	f1-score	support
glioma	0.54	0.75	0.63	324
meningioma	0.60	0.33	0.43	329
notumor	0.92	0.95	0.93	400
pituitary	0.88	0.91	0.90	352
accuracy			0.75	1405
macro avg	0.74	0.74	0.72	1405
weighted avg	0.75	0.75	0.74	1405

Figure 13: Classification Report of the custom VGG16

After inspecting the learning curves of the models I notice that when the weights are initialized to random the learning happens slower and the curve plateaus with great large fluctuations and frequently too. As seen in the figure below:

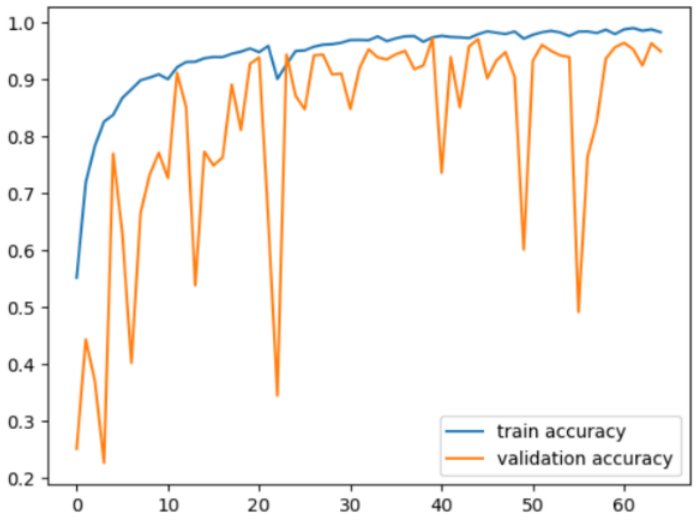

Figure 14: Accuracy curve of Resnet50_non_tf

The intense fluctuation in both the validation accuracy and loss curves implies that the model is struggling to generalize to new data. This means that while the model may perform well on the training data, it is not able to generalize its learned patterns to new, unseen data. The large fluctuations suggest that the model may be overfitting to the training data, which occurs when the model is too complex relative to

the amount of training data available. To address this issue, one can try reducing the complexity of the model or increasing the amount of training data. Doing more types of data augmentation than just the 3. Additionally, techniques such as regularization can also help prevent overfitting by adding penalties to the model's parameters during training to discourage complex patterns that may only fit the training data. Though regularization was used during training for these models. Overall, it is important to strike a balance between model complexity and the amount of available data to ensure the model can generalize well to new data.

4.2.3 Visualizing LIME interpretation of the classifiers with Heat Maps.

Using the libraries Matplotlib, scikit-image, and Lime. The code is used to generate and display Lime explanations for different pre-trained models. First the input image is defined and pre-processed. The Lime explainer is then defined, and a function to get the Lime explanation for a given model is created. The next section of the code calls the function for each of the pre-trained models and gets the Lime explanation. The predictions made by each model are also displayed. Finally, the code displays the original image and the Lime explanations for all the models using Matplotlib. [7] Figure 15 displays the output. An image with the original label Pituitary is used as the input image, which is then classified 9 times and their heat maps are displayed. The LIME generated images highlight the ten superpixels with the greatest impact, with green segments indicating the top superpixels that have a high probability of belonging to the predicted class, while the red segments represent the top superpixels that have a low probability of belonging to the predicted class.

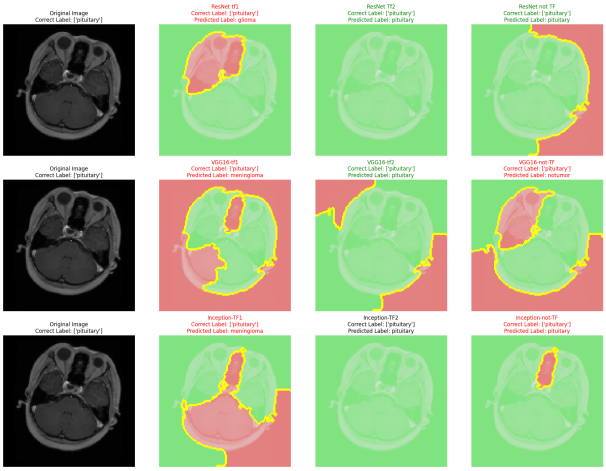

Figure 15: LIME Output

Unlike Smyth [8], who could get a little sense to why certain images were being incorrectly classified, I couldn't get similar heatmaps. Besides a few basic observations, there was nothing major I could conclude from the LIME interpretations. I just made these observations. Notice that for VGG16 according to our results, non_tf performed the worst because the network was too shallow to learn without the pre-trained knowledge and hence it incorrectly identified the pituitary as meningiaoma. The green segments partially cover part of the tumor and the red cover the other part. For a stronger model with correct classifications, there are more superpixels highlighted green covering more area. We cannot get a hit of how the localization is happening here.

5 Conclusion and Future Directions

In conclusion, the experiments and results presented in this report provide valuable insights into the use of pre-trained deep learning models for image classification tasks, specifically for small datasets with low similarity to the source dataset. The discussion of pre-training techniques highlights the importance of choosing the appropriate approach based on the size and similarity of the target dataset to the source dataset. The findings suggest that for a small dataset with low similarity to the source dataset, freezing the convolutional base of the pre-trained model and only training the added fully connected layers (tf1) may be the best approach, as it allows the model to benefit from the pre-trained weights while still adapting to the new data. This is consistent with previous literature on the topic. However, the results also suggest that the optimal approach may depend on the specific characteristics of the target dataset and the architecture of the pre-trained model. The performance of the different models varied across the three training types, and the performance of the models was affected by the number of trainable parameters and the number of layers in the architecture. For example, the Inception architecture, which is designed to be computationally efficient, performed better than the ResNet50 and VGG16 models on the small dataset with low similarity to the source dataset. Moreover, the impact of the number of layers on model performance was highlighted, as seen in the poor performance of VGG16 on the small dataset. Adding multiple sequential and functional layers on top of VGG16 significantly improved the performance, demonstrating the importance of appropriate architectural design and hyper-parameter tuning for optimal performance. The findings of this report demonstrate the importance of careful consideration of pre-training techniques and architectural design when using deep learning models for image classification tasks on small datasets with low similarity to the source dataset. The insights gained from this study can inform future research and practical applications of deep learning in various fields.

There are several directions for future research in this area. One possible avenue is to explore other pre-trained models and investigate their performance on small datasets with low similarity to the source dataset. Another direction is to experiment with different hyperparameters and training techniques to optimize the performance of the models. Additionally, incorporating more data augmentation techniques other than "rotation_range=15, horizontal_flip=True, vertical_flip=True" and ensembling could also potentially improve the performance of some models. Finally, it would be interesting to investigate the transferability of the models to other domains and tasks beyond image classification.

6 Refrences

[1] R. Mehrotra, M. A. Ansari, R. Agrawal, and R. S. Anand, "A Transfer Learning approach for AI-based classification of brain tumors," Machine Learning with Applications, vol. 2, p. 100003, Dec. 2020.

[2] S. Deepak and P. M. Ameer, "Brain tumor classification using deep CNN features via transfer learning," Computers in Biology and Medicine, vol. 111, p. 103345, Aug. 2019.

[3] M. Sajjad, S. Khan, K. Muhammad, W. Wu, A. Ullah, and S. W. Baik, "Multi-grade brain tumor classification using deep CNN with extensive data augmentation," Journal of Computational Science, vol. 30, pp. 174–182.

[4] L. Gaur, M. Bhandari, T. Razdan, S. Mallik, and Z. Zhao, "Explanation-Driven Deep Learning Model for Prediction of Brain Tumour Status Using MRI Image Data," Frontiers in Genetics, vol. 13, 2022.

[5] "Keras Documentation: Keras Applications API - Keras Documentation," [Online]. Available: https://keras.io/api/applications/vgg/.

[6]"Kaggle Documentation: Brain Tumor MRI Dataset" [Online]. Available https://www.kaggle.com/datasets/masoudnickparvar/brain-tumor-mri-dataset

[7] A. Sharma, "AbhiDS91/5770-Machine-Learning." [Online]. Available: https://github.com/AbhiDS91/5770-Machine-Learning

[8] B. Smyth, "Using LIME to explore Brain Tumor MRI Classification Model," Medium, Feb. 01, 2022. https://medium.com/@brookesmyth/using-lime-to-explore-brain-tumor-mri-classification-model-33d0a5b91e2c

[9] "Transfer Learning Pretrained Models in Deep Learning." https://www.analyticsvidhya.com/blog/2017/06/transfer-learning-the-art-of-fine-tuning-a-pre-trained-model/

[10] W. ML, "Brain Tumor Classification from MRI images using ResNet 50 Model." [Online]. Available: www.github.com/wisdomml2020/brain_tumour_classification/blob/main/Brain_Tumor_classification.ipynb

[11] B. Smyth, "Brain Tumor Image Classification." [Online]. Available: https://github.com/brooke57/BrainTumorImageClassification/blob/main/Exploring_the_Black_Box_with_Lime.ipynb