TABLE OF CONTENTS

ABSTRACT

The realm of medical imaging diagnosis, especially concerning brain tumor classification, is undergoing a transformative phase with the integration of artificial intelligence (AI) and machine learning (ML). The use of neural networks and deep learning algorithms has shown promising outcomes in streamlining the detection, characterization, and prediction of brain tumors from imaging data. Nevertheless, the effectiveness of these models is contingent on the availability of vast and varied datasets. The scarcity of adequately annotated brain tumor images continues to be a bottleneck in the development and performance of these classification models.

This research delves into the realm of Generative Adversarial Networks (GANs), specifically conditional GANs, to counteract the issue of data paucity within brain tumor imaging datasets. GANs, well-known for their capacity to generate new synthetic data, serve as a potential solution to expand the breadth and diversity of available training samples. Through the creation of artificial images based on existing data distributions, the primary aim is to bolster the dataset's richness, thus enabling more robust learning within brain tumor classification models.

This study endeavors to assess the effectiveness of incorporating artificially generated images in improving the classification models' performance. This investigation will encompass an in-depth comparative analysis, evaluating the classification model's accuracy, sensitivity, specificity, and robustness before and after the addition of synthetic images. The study will provide insights into the impact of synthetic data augmentation on the classification model's generalizability and performance, thereby contributing to the advancements in brain tumor classification techniques.

CHAPTER 1: INTRODUCTION

The progress in medical imaging, especially in the domain of brain tumor diagnosis, has been significantly propelled by artificial intelligence (AI) and machine learning (ML) algorithms. These technologies have substantially contributed to early tumor detection, precise localization, and accurate classification of brain cancers. However, the efficiency and reliability of these models are intrinsically reliant on the quality, size, and diversity of the datasets used for training. Despite the remarkable strides in AI-driven medical diagnostics, the inadequacy of annotated and diverse brain tumor images remains a persistent challenge.

1.1 Objective

The core objective of this study is to explore the capacity of conditional Generative Adversarial Networks (GANs) in addressing the limitations posed by insufficient datasets. GANs, renowned for their ability to generate new synthetic data, will be harnessed to augment the existing image repository for brain tumor classification. By synthetically creating images that align with the distributions present in the original dataset, the primary objective is to enrich the dataset's diversity and representation, thereby fortifying the learning capacity of brain tumor classification models.

Furthermore, this research will delve into the domain of transfer learning, specifically the utilization of pre-trained models, which will undergo fine-tuning processes both before and after the dataset augmentation. Transfer learning involves employing models pre-trained on extensive datasets and fine-tuning them on specialized tasks. In this study, pre-trained models will be employed in brain tumor classification tasks, first on the original dataset and subsequently on the dataset supplemented with images generated by conditional GANs. This approach aims to determine the impact of synthetic data augmentation on the performance and adaptability of pre-trained models, assessing changes in accuracy, sensitivity, specificity, and robustness before and after the integration of generated images.

By conducting a comparative analysis encompassing pre-trained model performance prior to and post dataset augmentation, this study seeks to unravel the effect of augmented data on enhancing the classification models' abilities in brain tumor diagnosis. The investigation aims to elucidate whether synthetic data generated by GANs can facilitate the improvement of classification models through enriching dataset diversity, contributing to advancements in accurate and robust brain tumor classification techniques.

CHAPTER 2: DATASET AND PREPROCESSING

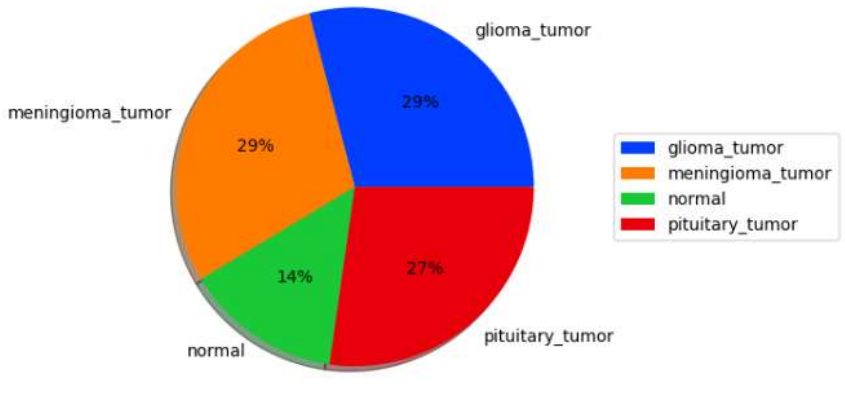
2.1 About the Data

The dataset used for this study comprises a total of 3096 brain MRI images categorized into four distinct classes: "Glioma Tumor," "Meningioma Tumor," "Pituitary Tumor," and "No Tumor Images."

The dataset is obtained from MRI scans collected across multiple medical institutions and public repositories. Each image is in grayscale and possesses a resolution of 256x256 pixels.

To facilitate model training, the dataset has been split into two subsets for the purpose of training. The training set consists of 80% of the images, and the validation set includes 20% of the total images.

The dataset is characterized by a balanced representation across different classes, allowing for comprehensive training and evaluation of the classification models. The images display varying characteristics and complexities, reflecting the challenges faced in real-world medical imaging datasets.



For the conditional GAN training, the images were first reduced from 256 x 256 pixels to 128 x 128 pixels. This was done because the Generative adversarial network is not strong enough to process or generate images of 256 pixels. Note that working on 256 x 256 pixel images gave me a ResourceExhaustedError which was then cleared when I reduced the images to 128 pixels.

I then saved the resized images into a new directory.

```
import os
import cv2

def load_images_and_resize(main_directory, new_directory, target_size):
    for class_folder in os.listdir(main_directory):
        class_path = os.path.join(main_directory, class_folder)
        if os.path.isdir(class_path):
            target_directory = os.path.join(new_directory, class_folder)
            os.makedirs(target_directory, exist_ok=True)

            for filename in os.listdir(class_path):
                img_dir = os.path.join(class_path, filename)
                try:
                    img = cv2.imread(img_dir)
                    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
                    img = cv2.resize(img, target_size)
                    cv2.imwrite(os.path.join(target_directory, filename), img)
                except Exception as e:
                    print(f"Error processing {filename}: {e}")


os.makedirs(new_directory, exist_ok=True)

target_size = (128, 128)
load_images_and_resize(main_directory, new_directory, target_size)
```

Furthermore, the images were scaled using a scaling function:

```
def scaling(x):
    x = (x-127.5)/127.5

    return x
```

In Conditional Generative Adversarial Networks (GANs), particularly the final layer of the generator network has a hyperbolic tangent (tanh) activation function. This scaling function is applied to the images for multiple reasons:

- The tanh function has an output range between -1 and 1. This activation function is commonly used in the final layer of a GAN's generator to ensure the generated images' pixel values also fall within the range of -1 to 1.

- The normalization of the image data to a range between -1 and 1 ensures consistency in the scale of the input images. This consistent scale is beneficial in training the GAN, as it aids in the convergence of the network and the stability of the training process.

CHAPTER 3: IMAGE CLASSIFICATION WITH CNNs:

3.1 What are Convolution Neural Networks?

Convolutional Neural Networks (CNNs) are a specialized type of artificial neural network primarily designed for processing structured grid-like data. They are particularly well-suited for tasks involving images and visual data. CNNs are composed of multiple layers that automatically detect and learn hierarchical patterns in the input data. Here's an overview of the main components and their functions:

1. Convolutional Layers: These layers perform feature extraction by applying a set of filters (kernels) to the input image. Each filter is small and slides across the image, computing dot products to identify local patterns. This process helps in capturing features like edges, textures, shapes, or more complex features in deeper layers.

2. Activation Functions: Non-linear functions like ReLU (Rectified Linear Unit) introduce non-linearity into the network. ReLU helps in enabling the network to learn more complex patterns by introducing non-linear transformations.

3. Pooling Layers: Pooling layers downsample the feature maps produced by convolution. Max pooling, for instance, selects the maximum value in a grid, effectively reducing the spatial dimensions of the feature maps while retaining the most salient information.

4. Fully Connected Layers: These layers process the information gathered from earlier layers and perform the classification or regression task. They connect every neuron in one layer to every neuron in the next layer, combining features learned from previous layers.

3.2 What is Transfer Learning?

Transfer learning is a machine learning technique where a model trained on one task is reused or repurposed to solve another related task. In this approach, knowledge (feature maps) acquired while solving the source task is leveraged to improve learning and performance on the target task.

Key elements of transfer learning:

1. Pre-trained Model: A model is pre-trained on a large dataset for a specific task, often a related problem domain. These pre-trained models are trained on large-scale datasets like ImageNet for image-related tasks.

2. Reuse of Knowledge: The knowledge and learned features captured by the pre-trained model can be applied to a new task by adapting, fine-tuning, or using these features as a foundation for learning on the new dataset.

3. Adjusting the Model: The model's final layers or intermediate layers are fine-tuned on the new dataset or task. The weights from earlier layers, often capturing general features like shapes or textures, are kept frozen or retrained to learn specific features related to the new task.

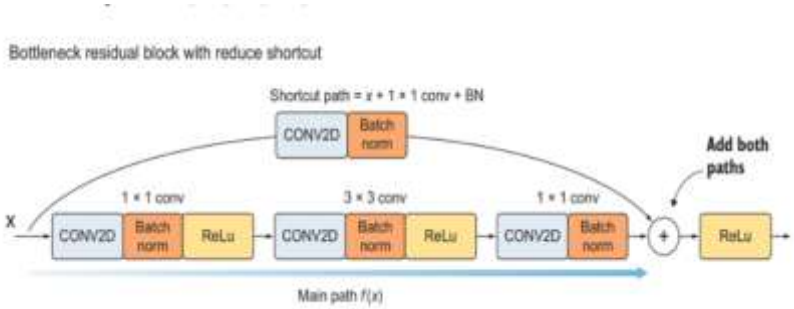
Transfer learning can significantly reduce the amount of labeled data required to train a model for a new task, accelerating the training process and improving its generalization. This is particularly useful when the target dataset is limited or lacks sufficient labeled samples. Additionally, it aids in mitigating issues such as overfitting.

For instance, a model pre-trained on a large dataset for image classification can be re-purposed for tasks like object detection or segmentation, requiring lesser effort and time for training compared to training a new model from scratch.

3.3: Our approach: The ResNet50 architecture

ResNet-50 is a deep convolutional neural network architecture that has 50 weight layers, making it relatively deep. It is part of the ResNet family, known for its innovative use of residual connections. These residual connections, commonly known as skip connections, enable the network to bypass certain layers by allowing the input from one layer to be added to the output of a deeper layer. This approach helps address the problem of vanishing gradients that typically occurs in training very deep networks.

The ResNet-50 architecture consists of several stages, and within each stage are multiple residual blocks. Each block contains convolutional layers, batch normalization, and an activation function. The significant feature of these blocks is the identity connection that skips one or more layers, allowing the gradient to flow more effectively during training. Each block typically contains two convolutional layers, which are followed by batch normalization and ReLU activation functions. The shortcut connections in these blocks aid in preserving and passing information through the network more efficiently.



The architecture of ResNet-50 allows it to achieve significant depth while mitigating the challenges associated with training such deep networks. With approximately 25.6 million parameters and 3.8 x 10^9 FLOPs, ResNet-50 has been widely employed in various computer vision tasks, including image classification, object detection, and feature extraction, demonstrating superior performance and accuracy in these applications.

3.4. What is Fine Tuning?

Fine-tuning pretrained models refers to the process of taking a pre-existing neural network that has been trained on a large dataset and adjusting or updating its parameters on a new, smaller dataset. This method is widely used in transfer learning, especially in scenarios where the new dataset has a different distribution or slightly different features than the original dataset used to train the pretrained model.

The process of fine-tuning involves unfreezing the parameters in the later layers of the pretrained model while keeping the initial layers frozen. By doing this, the latter layers can adapt and learn new, specific features or patterns from the new dataset. It is more common to modify or retrain the fully connected layers or the classifier of the model while keeping the convolutional base intact.

The technique benefits from the prior knowledge and learned features of the pretrained model. Since these models have been trained on large-scale datasets such as ImageNet, which contain diverse and general image features, they possess a broad range of learned representations that can be valuable for a wide array of computer vision tasks.

Fine-tuning allows the model to adjust its learned weights and features to the nuances of the new data, optimizing it for the specific problem it is being applied to. This method often reduces the training time and data requirements while achieving robust performance compared to training the model from scratch.

3.5 Appropriate level of transfer learning:

Choosing the level of transfer learning depends on two important factors:

1. Size of the target dataset (small or large): When we have a small dataset, the network probably won't learn much from training on larger number of layers, so it will tend to overfit the new data. In this case we want to rely more on the pretrained weights of the source dataset

2. Domain similarity of the source and target datasets: Is the target problem similar or different than the source problem ? These two factors lead to 4 different scenarios:

Scenario	Size of the target data	Similarity of the original and new datasets	Approach
1	Small	Similar	Use pretrained network as a feature extractor
2	Large	Similar	Freeze 60-80% of the network and fine tune the rest of the network
3	Small	Very Different	Since target dataset is different it might not be the best to freeze the higher level features of the pretrained network because they contain source dataset specific features. It will be better to retrain layers from somewhere early in the network. Freeze 30-50% of the network.
4	Large	Very Different	Fine tune the entire network/Train the entire network by using the existing weights of the pretrained network

3.6 Our classification Model:

1. *Taking our base model as ResNet50 and fine-tuning it:*

```
base_model =tf.keras.applications.resnet.ResNet50(include_top=False)
for layer in base_model.layers[:15]:
    layer.trainable=False
inputs = tf.keras.layers.Input(shape=(128, 128,3), name="input_layer")
```

We set up a ResNet50 model as the base model, excluding its fully connected layers (global average pooling and dense layers) by setting `include_top` to `False`. By doing so, the model retains its convolutional base for feature extraction while excluding the classifier part.

Next, we set the first 15 layers of the ResNet50 model as non-trainable. This freezes the weights and biases of these layers, preventing them from being updated during training. We have frozen 15 out of the architecture's 50 layers which is equivalent to freezing the first 30% of the model in order to use its basic pre-learned weights. This was done because the Brain Tumor MRI image dataset is vastly different from the Image Datagen on which ResNet50 was originally trained on. After trying with several different number of frozen layers, we settled on 15 as the ideal number as it gave us the best results.

A new input layer is defined with a shape of 128x128 and three channels (for a colored image) to match the input format of the original ResNet50 model. This input layer will be used for the modified architecture.

2. Building on top of the ResNet50 architecture:

```
#Average pool the outputs of the base model (aggregate all the most important information, reduce number of computations)
x = tf.keras.layers.GlobalAveragePooling2D(name="global_average_pooling_layer")(x)
x=tf.keras.layers.Flatten()(x)
x=tf.keras.layers.Dense(512, activation="relu")(x)
x=tf.keras.layers.Dropout(0.5)(x)
outputs=tf.keras.layers.Dense(4, activation="softmax",name="output_layer")(x)
model_1= tf.keras.Model(inputs, outputs, name="model")
```

1. Global Average Pooling Layer: The `GlobalAveragePooling2D` layer is added after the modified base model's output. This layer aggregates (averages) all feature maps or outputs of the previous layer to condense and reduce the number of parameters and computations, summarizing the most important information from the feature maps. It transforms the spatial data into a vector, which serves as the model's input for subsequent layers.

2. Flatten Layer: The `Flatten` layer reshapes the output of the Global Average Pooling layer into a 1D tensor. This flattening operation is necessary as subsequent dense layers (fully connected layers) require a one-dimensional input.

3. Dense Layer (Hidden): A densely connected neural network layer is added with 512 units (neurons) and ReLU activation. This layer helps in learning higher-level features from the aggregated features obtained from the previous layers.

4. Dropout Layer: The `Dropout` layer is introduced to prevent overfitting by randomly setting a fraction of input units to zero during training. It helps in improving the model's generalization by reducing interdependence on specific neurons during training.

5. Dense Layer (Output): Finally, the output layer is added with 4 units, using the softmax activation function. This is the classification layer responsible for producing the final probability distribution across the four classes (glioma tumor, meningioma tumor, pituitary

tumor, and no tumor). The softmax function ensures that the output values are in the range [0, 1] and their sum equals 1, representing class probabilities.

The resulting `model_1` is a complete neural network architecture, utilizing a pre-trained ResNet50 base model and additional custom-defined layers to perform classification for the brain tumor dataset across four classes.

CHAPTER 4: Conditional Generative Adversarial Networks.

4.1 An overview on Generative Adversarial Networks (GANs):

Generative Adversarial Networks, or GANs for short, are an approach to generative modeling using deep learning methods, such as convolutional neural networks.

Generative modeling is an unsupervised learning task in machine learning that involves automatically discovering and learning the regularities or patterns in input data in such a way that the model can be used to generate or output new examples that plausibly could have been drawn from the original dataset.

GANs are a clever way of training a generative model by framing the problem as a supervised learning problem with two sub-models: the generator model that we train to generate new examples, and the discriminator model that tries to classify examples as either real (from the domain) or fake (generated). The two models are trained together in a zero-sum game, adversarial, until the discriminator model is fooled about half the time, meaning the generator model is generating plausible examples.

GANs are an exciting and rapidly changing field, delivering on the promise of generative models in their ability to generate realistic examples across a range of problem domains, most notably in image-to-image translation tasks such as translating photos of summer to winter or day to night, and in generating photorealistic photos of objects, scenes, and people that even humans cannot tell are fake.

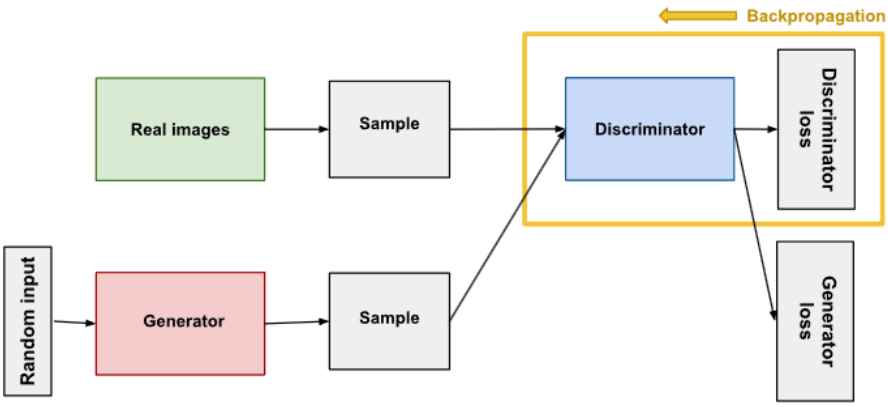
Note: Both the generator and the discriminator are neural networks. The generator output is connected directly to the discriminator input. Through backpropagation, the discriminator's classification provides a signal that the generator uses to update its weights.

4.1.1. The Discriminator:

The discriminator has two sources of training data

- **Real Data:** data of real objects, instances, people, etc. The discriminator treats these as positive examples during the training process.
- **Fake Data:** data samples created by the generator. These are regarded as negative examples during the training process.
-

During the training process the discriminator ignore the generator loss and just uses the discriminator loss. The discriminator loss penalizes itself for misclassification of real instance as fake or otherwise, followed by which it updates the weights of the discriminator network through backpropagation. An illustration is given below.



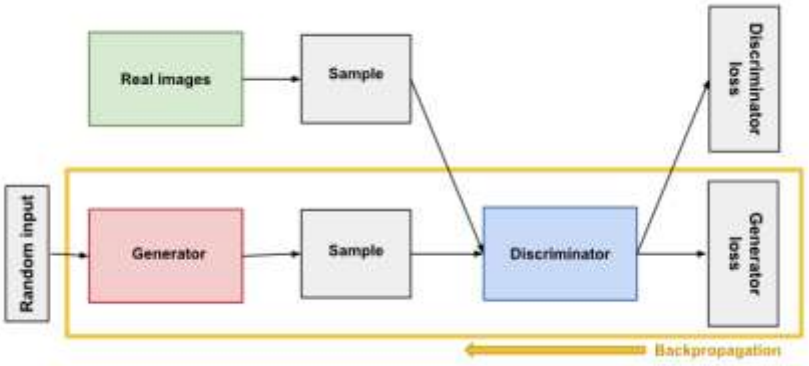
4.1.2. The Generator

As stated earlier the generator learns to create fake data by incorporating the discriminator's feedback. The components of the generator's training process include

- random input (some latent space vector representation)
- generator network
- discriminator network and its output
- generator loss which penalizes the generator for failing to fool the discriminator

Note: At the time of generator's training the discriminator does not update. Therefore the discriminator is only used her for its forward propagation output. The backpropagation performed on the discriminator is used to update the weights of the generator.

During the generator training process, some random noise is sampled as input, performing a forward propagation using the generator network and getting the discriminator to classify the generator output as Fake or Real. Followed by this the loss of the discriminator's classification is calculated and generator's loss in terms of failure of fooling the discriminator is calculated. The gradients obtained through backpropagation of both the discriminator and generator are then used to update the weights of the generator.



4.1.3 Minimax Loss:

$$E_x[log(D(x))] + E_z[log(1 - D(G(z)))]$$

In this function:

- D(x) is the discriminator's estimate of the probability that real data instance x is real.
- E_x is the expected value over all real data instances.
- G(z) is the generator's output when given noise z.
- D(G(z)) is the discriminator's estimate of the probability that a fake instance is real.
- E_z is the expected value over all random inputs to the generator (in effect, the expected value over all generated fake instances G(z)).
- The formula derives from the cross-entropy between the real and generated distributions.

The generator can't directly affect the log(D(x)) term in the function, so, for the generator, minimizing the loss is equivalent to minimizing log(1 - D(G(z))).

4.2 What are Conditional GANs?

Conditional GANs (cGANs) are an extension of traditional GANs that incorporate additional information, often in the form of labels or other conditioning variables, to control the data generation process. cGANs condition both the generator and discriminator networks on some extra information, allowing for the generation of more specific, controlled outputs.

The basic principle involves both the generator and the discriminator taking in additional information, along with the random noise fed into the generator. This extra information guides the generator to produce data that matches the given conditions.

The loss function for cGANs extends the original GAN loss by including the conditioning variable in both the generator and discriminator loss. The discriminator now not only aims to distinguish between real and fake data but also considers the conditional information. It aims to differentiate between real data samples with their respective conditions and the generated samples with the same conditions.

The C-GAN loss function is a combination of two components:

1. Adversarial Loss: Similar to the traditional GAN, the adversarial loss for the discriminator encourages it to distinguish between real data and generated data given the conditioning variable.

2. Conditional Loss: This loss ensures that the generated data satisfies the condition. It measures the difference between the generated data and the conditional information provided. This loss term is usually calculated using standard loss functions specific to the type of conditional information.

4.3 Our approach to Conditional GANs: Using the GANForge Library.

The GitHub library GANForge is a Python package designed for creating and training various types of Generative Adversarial Networks (GANs). It is built using TensorFlow and Keras, providing a convenient framework to implement and experiment with GAN architectures for image generation and other tasks.

The package allows users to quickly begin creating GANs by providing a simple code example. For instance, you can instantiate a GAN, set optimizers and loss functions, then fit it to your dataset. It supports several GAN architectures, including DCGAN, Conditional GAN, Super-Resolution GAN (SR GAN), among others. It also provides custom callbacks applicable to specific GANs, allowing for visualization or other custom functions during training.

The library aims to offer a range of GAN architectures and resources that can be leveraged for different types of image generation tasks, supporting researchers and developers working in the field of GANs. It's a tool for experimenting with and implementing various GAN architectures efficiently.

Taking a look at its discriminator and generator models:

```
def _create_discriminator(
    self,
    input_shape: Tuple[int, int, int]
) -> Model:
    input_label = Input(shape=(1,))
    x = Embedding(self.num_classes, 50)(input_label)
    x = Dense(input_shape[0] * input_shape[1])(x)
    label = Reshape((input_shape[0], input_shape[1], 1))(x)

    input_image = Input(shape=input_shape)
    concat = Concatenate()([input_image, label])

    x = Conv2D(32, kernel_size=5, strides=2, padding="same")(concat)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Conv2D(64, kernel_size=5, strides=2, padding="same")(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(0.2)(x)
    x = Conv2D(128, kernel_size=5, strides=2, padding="same")(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(0.2)(x)
    x = Conv2D(256, kernel_size=5, strides=2, padding="same")(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(0.2)(x)
    x = Conv2D(256, kernel_size=5, strides=2, padding="same")(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(0.2)(x)
    x = Flatten()(x)
    x = Dropout(0.4)(x)
    output = Dense(1, activation="sigmoid")(x)

    return Model(inputs=[input_image, input_label], outputs=[output])
```

```
def _create_generator(
    self,
    input_shape: Tuple[int, int, int],
    latent_dim: int,
    discriminator: Sequential
) -> Model:
    g_h = discriminator.layers[-7].output_shape[1]
    g_w = discriminator.layers[-7].output_shape[2]
    g_d = discriminator.layers[-7].output_shape[3]

    input_label = Input(shape=(1,))
    x = Embedding(self.num_classes, 50)(input_label)
    x = Dense(g_h * g_w)(x)
    label = Reshape((g_h, g_w, 1))(x)

    input_latent = Input(shape=(latent_dim,))
    x = Dense(g_h * g_w * g_d)(input_latent)
    x = Reshape((g_h, g_w, g_d))(x)
    concat = Concatenate()([x, label])
    x = Conv2DTranspose(256, kernel_size=4, strides=2, padding="same")(concat)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Conv2DTranspose(128, kernel_size=4, strides=2, padding="same")(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Conv2DTranspose(64, kernel_size=4, strides=2, padding="same")(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Conv2DTranspose(64, kernel_size=4, strides=2, padding="same")(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)
    output = Conv2D(input_shape[2], kernel_size=5, padding="same", activation="tanh")(x)

    return Model(inputs=[input_latent, input_label], outputs=[output])
```

4.4 Training our Conditional GAN

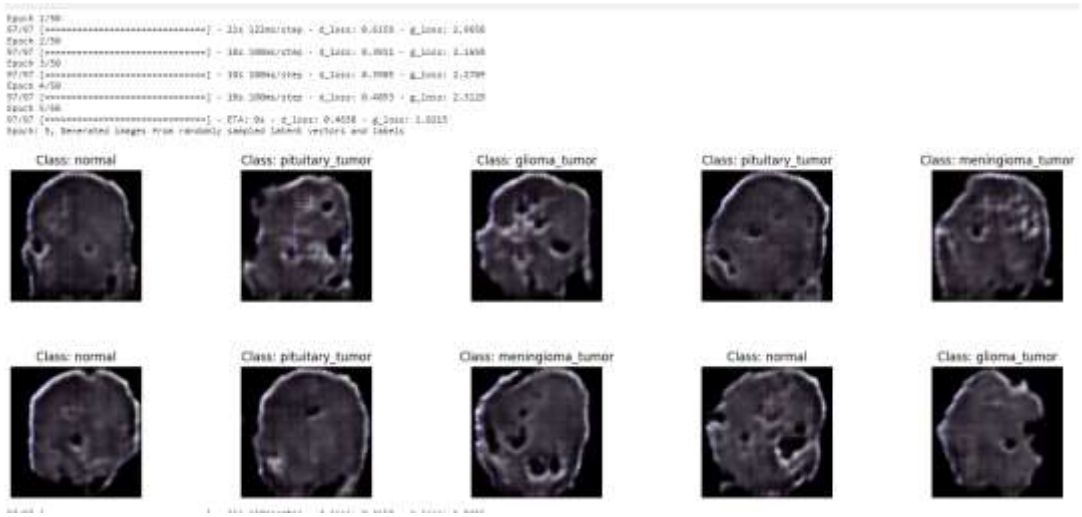
```
model_GAN = ConditionalDCGAN(input_shape=(img_size, img_size, 3), latent_dim=latent_dim, num_classes=4)
model_GAN.compile(d_optimizer=Adam(learning_rate=0.0002, beta_1=0.5),
                  g_optimizer=Adam(learning_rate=0.0002, beta_1=0.5),
                  loss_fn=BinaryCrossentropy())
```

We input the image size, latent vectors of noise for image generation and its corresponding class. d_optimizer and g_optimizer: Adam optimizers are specified for both the discriminator and generator parts of the cDCGAN. They use a learning rate of 0.0002 and a beta value of 0.5.

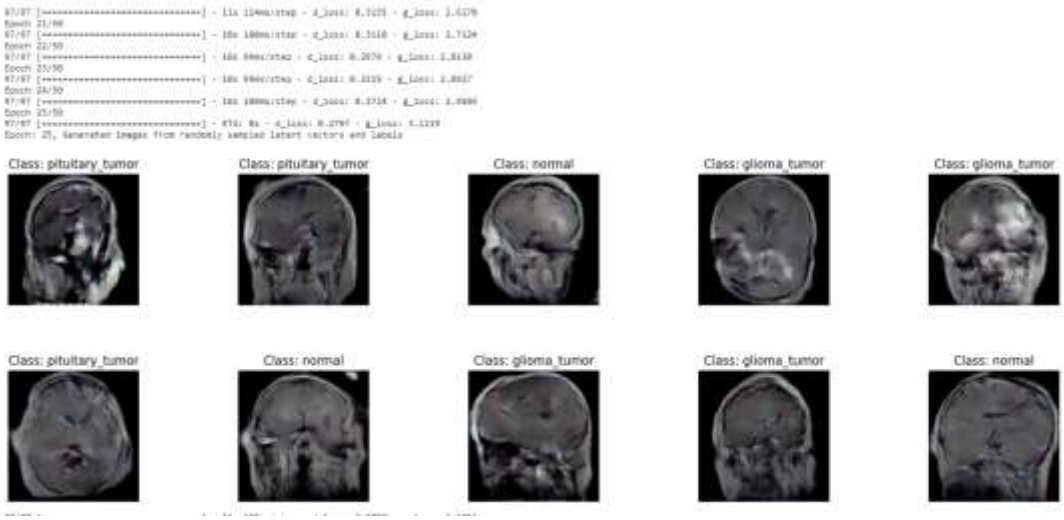
loss_fn: Binary cross-entropy is chosen as the loss function for the model. This loss function is commonly used in GANs to measure the difference between real and generated images.

We then train the model for 50 epochs.

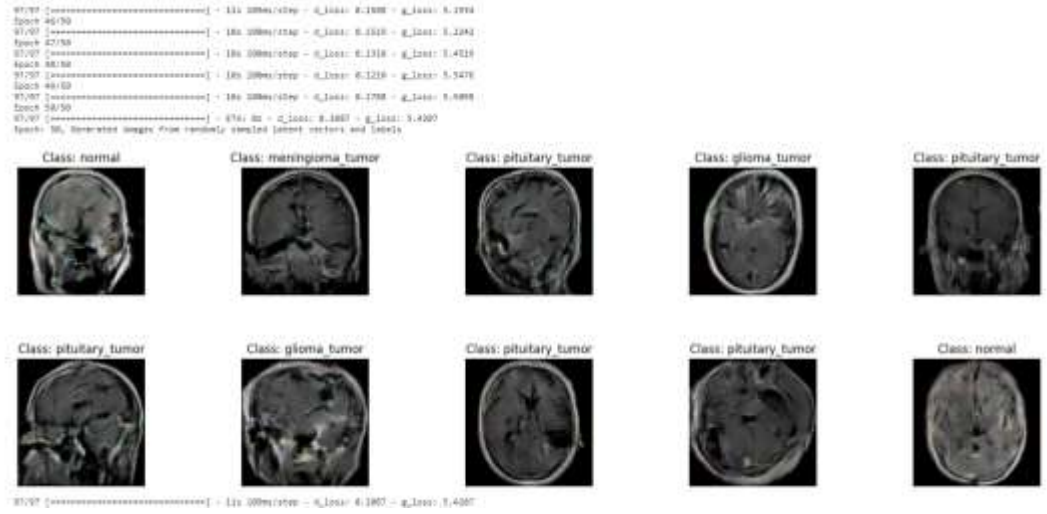
Output after 5 epochs:



25 epochs:



50 epochs:



4.5. Generating images:

We generated 10000 images, 2500 for each class.

```
num_samples = 10000  # Number of images you want to generate
latent_dim = 256  # Should match the latent_dim used during training
num_classes = 4  # Number of classes in your dataset

# Generate random latent vectors and labels
random_latent_vectors = np.random.normal(size=(num_samples, latent_dim))
random_labels = np.random.randint(0, num_classes, size=(num_samples, 1))

# Mapping random labels to tumor types using the mapping dictionary
mapped_labels = [label_mapping[label] for label in random_labels.flatten()]
generated_images = model_GAN.generator.predict([random_latent_vectors, random_labels])
```

- `random_latent_vectors`: Creates a set of random vectors that serve as input to the generator. The size matches the number of images to generate (`num_samples`).

- `random_labels`: Produces random labels (0 to `num_classes - 1`) to condition the image generation process for the GAN. These labels correspond to different tumor types in this scenario.

- `model_GAN.generator.predict`: Utilizes the generator network of the Conditional DCGAN model (`model_GAN`) to produce artificial images.It takes in both the random latent vectors (`random_latent_vectors`) and the generated label mappings (`mapped_labels`) to condition the image generation process. This generates images that correspond to the specified tumor types based on the labels.

CHAPTER 5: COMPARING CLASSIFICATION OUTPUTS

5.1. Classification before augmenting data with generated images:

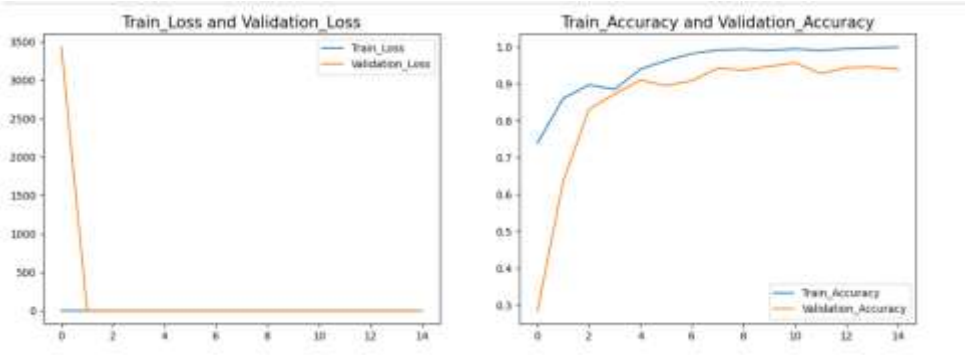
5.1.1. Training the model:

5.1.2. Last 5 epochs:

```
history_df= pd.DataFrame(history.history)
history_df.tail()
```

	loss	accuracy	val_loss	val_accuracy	lr
10	0.018003	0.994348	0.236121	0.956381	0.00050
11	0.030255	0.990311	0.340171	0.927302	0.00050
12	0.017906	0.994752	0.262804	0.943457	0.00050
13	0.009426	0.996770	0.265219	0.945073	0.00025
14	0.006161	0.998789	0.413693	0.938611	0.00025

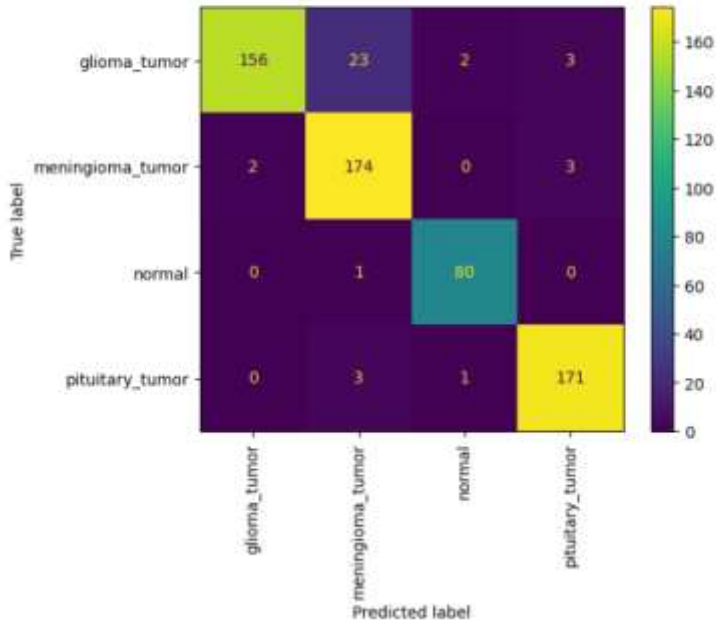
5.1.3 Comparing training and validation loss and accuracy:



5.1.4 Model Evaluation:

	precision	recall	f1-score	support
0	0.99	0.85	0.91	184
1	0.87	0.97	0.92	179
2	0.96	0.99	0.98	81
3	0.97	0.98	0.97	175
accuracy			0.94	619
macro avg	0.95	0.95	0.94	619
weighted avg	0.94	0.94	0.94	619

Confusion_Matrix



	ResNet50
Precision_Score	0.946173
Recall_Score	0.945743
F1_Score	0.943818

- Class 0 (Glioma): The model indicates a very high precision of 99%, which means when it predicts samples as Glioma, it is accurate 99% of the time. However, the recall is relatively lower at 85%, signifying that the model identifies 85% of the actual Glioma cases. The F1-score, a harmonic mean of precision and recall, stands at 91%. This might indicate a trade-off between precision and recall, but with an overall good balance.

- Class 1 (Meningioma): The model demonstrates an 87% precision and a high recall of 97%. This implies that while it is slightly less accurate at predicting Meningioma compared to Glioma, it captures 97% of the true Meningioma cases. The F1-score here is 92%.

- Class 2 (Normal): Shows an excellent performance with 96% precision and high recall at 99%. The F1-score is notably high at 98%, indicating a robust performance in identifying 'Normal' cases.

- Class 3 (Pituitary): The model yields a high precision of 97% and a high recall of 98%, resulting in a solid F1-score of 97%. This shows the model's ability to accurately predict and capture a significant portion of the Pituitary tumor cases.

- The macro and weighted averages demonstrate an overall performance, considering the different classes in the dataset. With an accuracy of 94%, the model showcases strong performance across all classes, particularly excelling in the accurate identification of normal cases, while also maintaining a good balance between precision and recall in identifying the tumor classes

5.2 Classification after augmenting data with generated images:

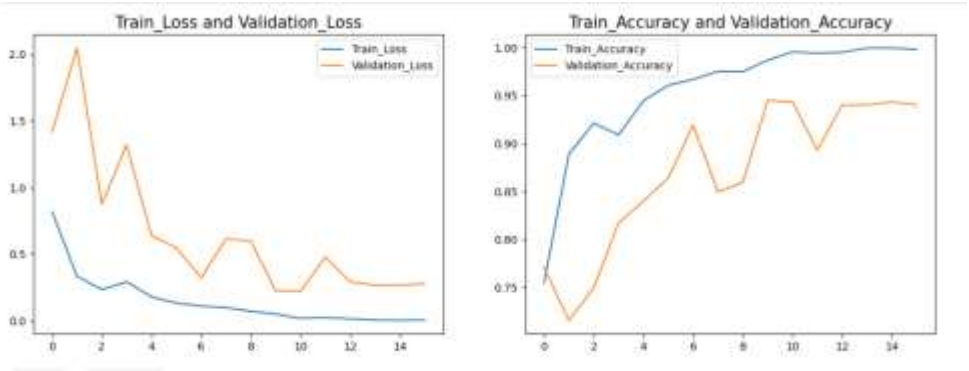
5.2.1. Training the model:

5.2.2. Last 5 epochs:

```
history_resnet50= pd.DataFrame(history2.history)
history_resnet50.tail()
```

	loss	accuracy	val_loss	val_accuracy	lr
11	0.021402	0.994348	0.477543	0.893376	0.000500
12	0.013503	0.995155	0.290341	0.940226	0.000500
13	0.003258	0.999596	0.266628	0.940226	0.000250
14	0.001557	0.999596	0.265674	0.943457	0.000250
15	0.003640	0.998385	0.276976	0.940226	0.000125

5.2.3 Comparing training and validation loss and accuracy:



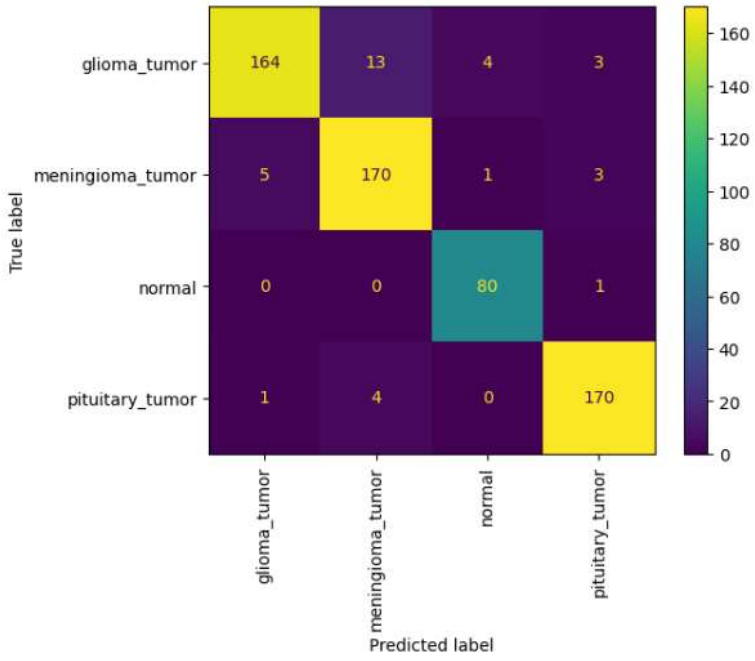
5.2.4 Model Evaluation:

```
-----------------------
              precision    recall  f1-score   support

           0       0.96      0.89      0.93       184
           1       0.91      0.95      0.93       179
           2       0.94      0.99      0.96        81
           3       0.96      0.97      0.97       175

    accuracy                           0.94       619
   macro avg       0.94      0.95      0.95       619
weighted avg       0.94      0.94      0.94       619

Confusion_Matrix
-----------------------
```



:

	ResNet50
Precision_Score	0.950027
F1_Score	0.946320
Recall_Score	0.943856

- Class 0 (Glioma): The precision is 96%, indicating that when the model predicts samples as Glioma, it is correct 96% of the time. The recall, which measures the ability to identify actual Glioma cases, is at 89%. The F1-score, the harmonic mean of precision and recall, is 93%.

- Class 1 (Meningioma):The precision for Meningioma is 91%, suggesting that 91% of the predictions for Meningioma are accurate. The recall is high at 95%, showing that the model captures 95% of the actual Meningioma cases. The F1-score stands at 93%.

- Class 2 (Normal): Shows strong precision at 94% and a very high recall of 99%, resulting in a high F1-score of 96%.

- Class 3 (Pituitary): A precision of 96% suggests that when predicting Pituitary tumors, it is accurate 96% of the time. The recall of 97% indicates that the model captures 97% of the true Pituitary tumor cases. The F1-score is 97%.

The macro and weighted averages consider the performance across all classes. The model demonstrates an accuracy of 94%, with a balance of strong precision and recall across the different tumor classes, particularly excelling in identifying Normal cases.

5.3 Inference

Comparing the two classification reports, both models seem to perform very similarly, with high accuracy and good precision, recall, and F1-scores across the different tumor types.

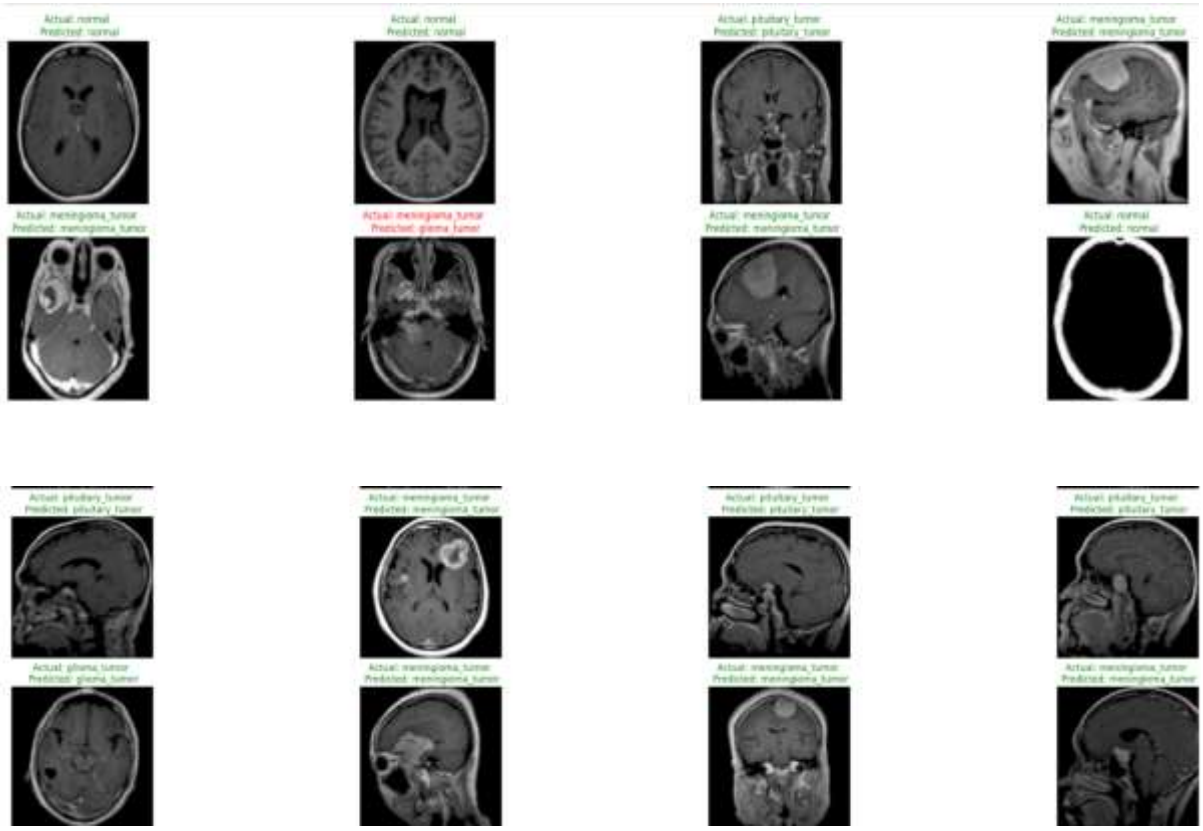
1. First Model:
 - Glioma: Precision 99%, Recall 85%, F1-score 91%
 - Meningioma: Precision 87%, Recall 97%, F1-score 92%
 - Normal: Precision 96%, Recall 99%, F1-score 98%
 - Pituitary: Precision 97%, Recall 98%, F1-score 97%
 - Overall accuracy: 94%

-2. Second Model:
 - Glioma: Precision 96%, Recall 89%, F1-score 93%
 - Meningioma: Precision 91%, Recall 95%, F1-score 93%
 - Normal: Precision 94%, Recall 99%, F1-score 96%
 - Pituitary: Precision 96%, Recall 97%, F1-score 97%
 - Overall accuracy: 94%

Both models demonstrate excellent performance, but the second model with the augmented data appears to have slightly higher recall for Glioma and Meningioma, which might suggest a slightly better ability to identify those tumor types accurately.

5.3.1 Deeper evaluation of the classification output on augmented data:

- Visualising predicted and actual images:



- Why was there no major difference in the evaluation metrics?

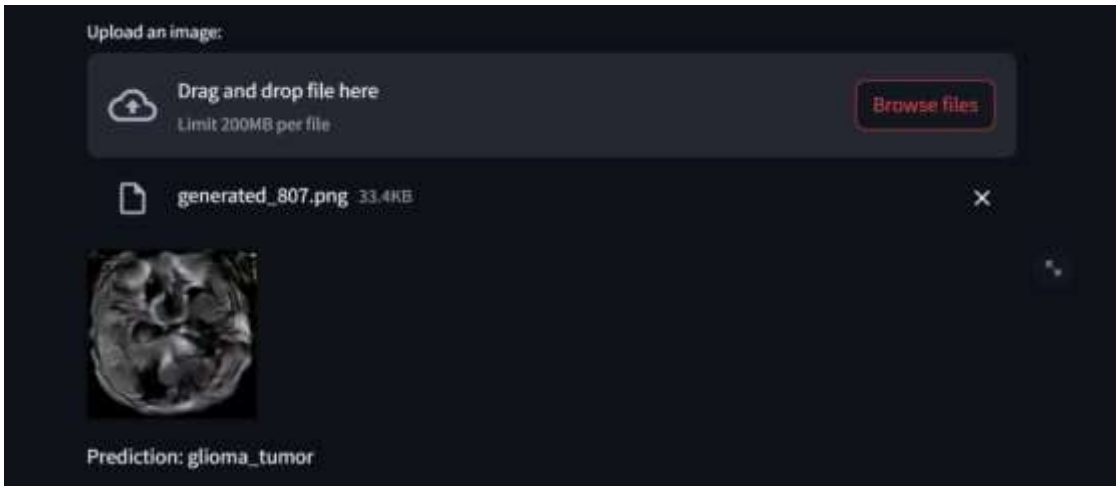
    - While we see that the classification on the data augmented by the images generated by GANs performs better, the difference is not too significant. This could be because the C-GAN model was not strong enough to fully capture the complexities seen in black and white MRI Scans and thus the images generated were not as distinctive as the original data.

CHAPTER 6: STREAMLIT DEPLOYMENT

Streamlit is an open-source Python library used for creating web applications for machine learning and data science projects. It provides a simple and intuitive way to convert data scripts into shareable web apps. Streamlit allows developers to visualize data, create interactive web apps, and deploy machine learning models with ease.

The deployment and launch of the classification model was not possible since the model was of 295 MB and streamlit only allows upto 25 MB.

On locally running the streamlit app, we get the following results:



Both of these test images were correctly classified

Chapter 7: Conclusion and Future Scope

While the application of Generative Adversarial Networks (GANs) in creating synthetic data for brain tumor classification showed potential, there were limitations in the quality of the generated dataset. As a result, the enhancement of the Convolutional Neural Networks (CNNs) used for classification was not as substantial as expected.

The GAN's inability to produce high-quality synthetic images restrained the extent to which the CNN model could be improved. Consequently, the enhancements in brain tumor classification using CNNs were only marginally successful.

Future Prospects:

- *Enhanced GAN Architecture*: There's a promising avenue for developing improved GAN architectures tailored for medical imaging tasks. By refining GANs to generate more realistic and high-quality images, these models can potentially provide more effective data augmentation for classification.

- *Advanced Image Generation:* Exploring state-of-the-art GAN architectures and innovative methodologies for GAN-based data augmentation could yield significantly improved results for CNN classification models.

- *Data Quality and Quantity*: Emphasizing the acquisition of larger, higher-quality datasets to provide GANs with more diverse and robust samples for training classification models.

- *Adaptable CNN Architectures*: Experimenting with more advanced and adaptable CNN architectures to effectively leverage augmented datasets, potentially leading to significant improvements in brain tumor classification accuracy.

Continued research and development in GAN architectures and innovative strategies for data augmentation will play a critical role in improving the quality of synthetic data for medical image classification. These advancements are key to enabling more precise and reliable diagnostic procedures for brain tumor analysis.

REFERENCES

Papers:

- [CGAN Paper](https://arxiv.org/abs/1411.1784): Paper on Conditional Generative Adversarial Networks.

Tutorials:

- [CGAN Explanation on Towards Data Science](https://towardsdatascience.com/cgan-conditional-generative-adversarial-network-how-to-gain-control-over-gan-outputs-b30620bd0cc8): Detailed explanation of CGANs and their application.

Example Code, Library:

- [GANForge GitHub](https://github.com/quadeer15sh/GANForge/tree/main): Repository containing code for GANForge.

- [Scikit-learn Digits Classification Example](https://scikit-learn.org/stable/auto_examples/classification/plot_digits_classification.html): Example code for digit classification using scikit-learn.

Additional Resources:

- [GAN Structure - Google Developers](https://developers.google.com/machine-learning/gan/gan_structure): Google Developers documentation on GAN structure.

- [Machine Learning Mastery - Conditional GAN Tutorial](https://machinelearningmastery.com/how-to-develop-a-conditional-generative-adversarial-network-from-scratch/): Tutorial on developing CGANs from scratch.

- [Inception Score Paper](https://arxiv.org/abs/1406.2661): Paper on the Inception Score metric.

- [Google Developers - GAN Structure](https://developers.google.com/machine-learning/gan/gan_structure): Detailed information on GAN structure and architecture.