Implementation of a Model for Detection and Classification of Brain Tumours in Magnetic Resonance Imaging using Convolutional Neural Networks

PROYECTO

presentado para optar

al Título de Grado en Ingeniería Biomédica por

Álvaro Serra i Parri

bajo la supervisión de

Javier Díaz Dorronsoro

Donostia-San Sebastián, julio 2023

tecnun
Universidad de Navarra

1. Acknowledgments:

A mis padres, Francisco y María Dolores, por forjar mi ser y darme la oportunidad de ser.

A mis hermanos, María, Fernando, Javier, Laura y Beatriz, por ser faros en la moralidad, el aprendizaje y la formación de mis virtudes.

A Marta, por haber sido mi ciudad amurallada durante este mes.

A "Los Últimos de Zayas", por otorgarle un sentido positivo a la melancolía.

A Jesús, por hacer de guía por mi mar académico.

Al Colegio Mayor Ayete, por haber sido condición *sine qua non* a mi crecimiento personal y académico.

Y a Dios y a la Virgen María, por las bendiciones derramadas, por las sendas trazadas y las manos tendidas en mi caminar.

Table of Contents

2. Abstract:

Accurate detection and classification of brain tumours in magnetic resonance imaging (MRI) are crucial for diagnosis and treatment planning. This research paper presents the implementation of a comprehensive model for the detection and classification of brain tumours using convolutional neural networks (CNNs) based on T1-weighted MRI scans.

The project encompasses the development of a data preprocessing pipeline, including data normalisation, train/validation/test set splitting, and organisation into a suitable directory structure. The pipeline ensures the creation of a balanced and representative dataset for training and evaluating the CNN-based tumour classification model.

The tumour detection and classification algorithm utilize CNNs to analyse preprocessed T1-weighted MRI data. The 3D CNN model leverages the spatial information encoded in the MRI volumes to accurately identify and classify brain tumours. TensorFlow, a popular deep learning library, is employed for developing and training the 3D CNN model.

The model's performance is evaluated using appropriate metrics such as accuracy, precision, and area under the ROC curve (AUC). The results demonstrate the effectiveness of the proposed model in detecting and classifying brain tumours in T1-weighted MRI scans, with high accuracy and discriminatory power.

Overall, the implementation of this model for brain tumour detection and classification in T1-weighted MRI scans provides a valuable tool for medical professionals and researchers. The model's accuracy and efficiency contribute to improved diagnosis, treatment planning, and monitoring of brain tumours, ultimately enhancing patient care and outcomes.

3. Introduction:

3.1. Background:

Glioblastoma, also known as glioblastoma multiforme (GBM), is the most aggressive and common form of primary brain tumours. It accounts for a significant proportion of malignant brain tumours in adults and poses considerable challenges in terms of diagnosis, treatment, and patient prognosis. According to the American Brain Tumor Association, glioblastoma represents approximately 15% of all primary brain tumours, with an incidence rate of 3.19 cases per 100,000 population [1]. This high prevalence underscores the urgent need for accurate detection and classification methods.

The accurate classification of glioblastoma plays a crucial role in guiding treatment strategies and predicting patient outcomes. Traditional classification methods, such as histopathological analysis, have inherent limitations that impact diagnosis and treatment decisions. Histopathology relies on subjective interpretations and is susceptible to interobserver variability, making it time-consuming and potentially inaccurate Smith A, et al. (2019) [8]. For instance, a study conducted by Smith C [9], et al. (2020) revealed a substantial discrepancy in tumour grading between pathologists, leading to inconsistencies in treatment plans and prognoses. These limitations hinder the ability to provide timely and appropriate interventions for patients.

In recent years, there has been a growing interest in leveraging machine learning and artificial intelligence techniques to improve the accuracy and efficiency of glioblastoma classification. Machine learning algorithms, particularly convolutional neural networks (CNNs), have shown promise in various medical imaging tasks, including brain tumour detection and classification. These algorithms can learn intricate patterns and features directly from medical imaging data, enabling more objective and reliable classification outcomes.

However, before applying machine learning algorithms, appropriate preprocessing of the data is essential. Preprocessing involves various steps, such as data normalisation, feature extraction, and dataset splitting. These steps aim to enhance the quality, consistency, and suitability of the data for subsequent analysis. In the context of glioblastoma classification, preprocessing plays a vital role in improving the performance and reliability of machine learning models.

In this research project, we focus on developing a comprehensive data preprocessing workflow specifically tailored for glioblastoma classification. The objective is to preprocess the raw glioblastoma imaging data, extract relevant features, and create balanced and representative datasets for training, validation, and testing. By optimizing the preprocessing workflow, we aim to improve the accuracy, robustness, and generalisation capabilities of glioblastoma classification models.

The development of an effective data preprocessing workflow for glioblastoma classification has the potential to significantly impact clinical decision-making, patient stratification, and treatment planning. It can provide clinicians and researchers with valuable insights into the underlying characteristics and patterns of glioblastoma tumours, leading to improved diagnostic accuracy and personalised therapeutic interventions.

By addressing the challenges associated with glioblastoma classification and optimizing the preprocessing workflow, we aim to contribute to the advancement of glioblastoma research and ultimately improve patient outcomes.

3.2. Research objectives:

The main objectives of this research project are to:

1. Develop a comprehensive data preprocessing workflow specifically tailored for glioblastoma classification. This includes implementing data normalisation

techniques, feature extraction methods, and data augmentation strategies to enhance the quality and relevance of the input data.

2. Investigate the impact of different preprocessing techniques on the performance and accuracy of glioblastoma classification models. By systematically evaluating various preprocessing steps and combinations, we aim to identify the most effective preprocessing strategies that contribute to improved classification outcomes.
3. Create balanced and representative datasets for training, validation, and testing. This involves carefully selecting and partitioning the available glioblastoma data to ensure a fair distribution of samples from different classes and minimise bias during model training and evaluation.
4. Explore and compare different machine learning algorithms and techniques for glioblastoma classification. By considering a range of approaches, including traditional machine learning algorithms and deep learning architectures, we aim to identify the most suitable methods for accurately classifying glioblastoma tumours.
5. Evaluate the performance and generalisation capabilities of the developed classification models. This includes assessing the accuracy, sensitivity, specificity, and other performance metrics on independent test datasets to gauge the reliability and applicability of the models in real-world clinical settings.
6. Assess the interpretability of the classification models using visualisation techniques and feature importance analysis. By employing these methods, we aim to gain insights into the underlying features and characteristics contributing to glioblastoma classification, facilitating clinical acceptance and enhancing our understanding of the disease.
7. Contribute to the advancement of glioblastoma research by providing insights into the potential use of machine learning techniques in improving diagnostic accuracy,

treatment planning, and patient stratification. By highlighting the strengths and limitations of the developed models, we aim to pave the way for future research and clinical applications.

Through these research objectives, we aim to address the challenges associated with glioblastoma classification and contribute to the field by developing an effective preprocessing workflow and accurate classification models. Ultimately, our goal is to improve the understanding, diagnosis, and treatment of glioblastoma, leading to better patient outcomes and quality of life.

3.3. Motivation:

The motivation behind this research project stems from the urgent need to develop robust and automated approaches for glioblastoma classification. Glioblastoma, as a highly aggressive and devastating form of brain cancer, presents significant challenges in diagnosis and treatment. Traditional diagnostic methods, which rely on manual interpretation and subjective assessments, often introduce variability and limit the accuracy of glioblastoma classification.

To emphasize the importance of our research, I draw upon a personal anecdote that underscores the real-world implications of accurate diagnosis. During a visit to the BCBL (Basque Center on Cognition, Brain and Language), I volunteered for a brain imaging procedure using an MRI machine. The images revealed an anomaly in my brain that was initially suspected to be a tumour. This experience ignited a profound curiosity in me and highlighted the critical role of accurate diagnostic tools in the field of neuroscience.

By blending personal experience with the broader motivation, we emphasize the relevance and significance of developing an automated brain tumour classification system using machine learning techniques. Our aim is to enhance the accuracy and efficiency of

glioblastoma diagnosis, leading to improved treatment planning and patient care. Furthermore, we recognise that the complexity of glioblastoma demands a comprehensive understanding of tumour characteristics and imaging features, which can be achieved through the interpretability of classification models.

Through this research, we seek to provide clinicians with powerful tools for early detection, personalised treatment strategies, and better patient management. The impact of reliable and accurate classification models extends beyond medical imaging and oncology, resonating with the broader healthcare community. By addressing critical healthcare challenges and making tangible advancements in glioblastoma classification, we aim to improve patient outcomes and contribute to the field of medical imaging, machine learning, and cancer research.

By weaving the personal motivation into the broader context, we highlight the dedication to utilising advanced technologies to make a significant difference in the lives of individuals affected by glioblastoma. This personal commitment further reinforces the importance of the research project and adds a compelling element to the overall narrative.

3.4. Literature review:

Glioblastoma classification is a critical task in improving diagnosis and treatment strategies for this aggressive form of brain cancer. Numerous studies have focused on developing robust approaches using machine learning and image analysis techniques to enhance the accuracy and efficiency of glioblastoma classification. In this literature review, we will delve into the existing research and identify key findings, methodologies, and limitations in the field.

One essential area of investigation is the utilisation of advanced imaging modalities, such as magnetic resonance imaging (MRI), for glioblastoma classification. Ellingson et al.

(2014) [3] proposed a standardised brain tumour imaging protocol for clinical trials, which has become a valuable resource for acquiring consistent and high-quality imaging data. Additionally, Chang et al. (2017) [2] emphasized the potential of deep learning methods in radiology, highlighting their ability to extract intricate features and improve diagnostic accuracy.

Deep learning approaches, specifically convolutional neural networks (CNNs), have gained prominence in medical image analysis, including glioblastoma classification. Havaei et al. (2017) [5] presented a CNN-based brain tumour segmentation framework, demonstrating its efficacy in accurate tumour delineation. The Multimodal Brain Tumour Image Segmentation Benchmark (BRATS) by Menze et al. (2015) [7] has facilitated the evaluation and comparison of various segmentation algorithms, enabling advancements in this field.

While these studies have made significant strides, certain gaps and limitations remain. One challenge is the scarcity of labelled training data due to the rarity and complexity of glioblastoma cases. This limitation affects the generalizability and performance of classification models. Furthermore, the interpretability of deep learning models in glioblastoma classification is often a concern. Understanding the discriminative factors and features that contribute to the classification decisions is crucial for clinical acceptance and furthering our knowledge of the disease (Lao et al., 2017) [6].

To address these gaps, our research project aims to contribute to the field of glioblastoma classification by exploring novel data preprocessing techniques and developing interpretable deep learning models. By investigating feature extraction methods and incorporating domain knowledge, we seek to enhance the quality and representativeness of

the training data. The utilisation of explainable AI techniques will enable clinicians and researchers to gain insights into the decision-making process of the classification models.

Moreover, we recognise the importance of considering multi-modal imaging data, such as incorporating functional imaging or molecular information, to improve the accuracy of glioblastoma classification (Gutman et al., 2013) [4]. By integrating diverse imaging modalities, we can capture a more comprehensive understanding of tumour characteristics and potentially uncover new biomarkers for diagnosis and treatment planning.

In summary, this literature review highlights the advancements made in glioblastoma classification using machine learning and image analysis techniques. It acknowledges the gaps and limitations in the existing approaches, particularly in terms of data availability and model interpretability. Our research project aims to bridge these gaps by investigating novel data preprocessing techniques, developing interpretable deep learning models, and exploring the potential of multi-modal imaging data.

By addressing these challenges, we aspire to contribute to the growing body of knowledge in glioblastoma classification, enhance clinical decision-making, and ultimately improve patient outcomes. Through our research endeavours, we aim to inspire further advancements in medical imaging, machine learning, and cancer research, bringing us closer to combating the devastating impact of glioblastoma.

4. Methodology

4.1. Dataset Description:

For this research project, two distinct datasets were utilised: the glioblastoma dataset and the healthy control dataset. These datasets were selected based on their relevance to the study objectives and the availability of comprehensive imaging data.

The glioblastoma dataset was obtained from The Cancer Imaging Archive (TCIA). It comprises a collection of MRI scans from patients diagnosed with glioblastoma, providing a diverse representation of glioblastoma cases from multiple centres.

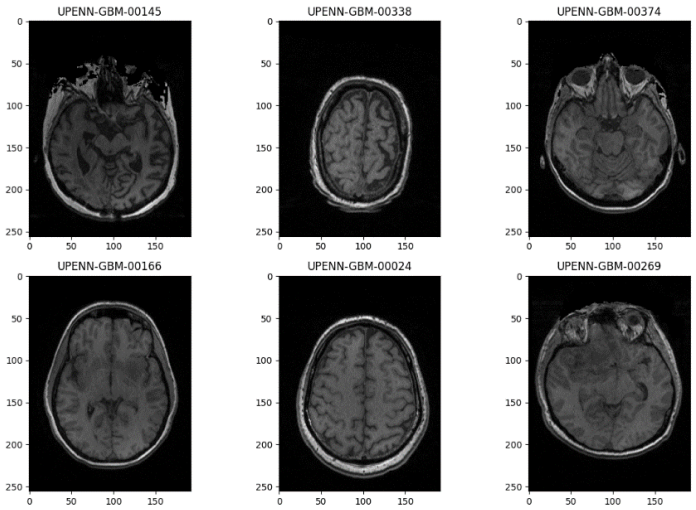

Figure 1: Random Slices from Random Tumorous Patients

The healthy control dataset used in this study was sourced from the IXI Dataset provided by the Imperial College. This dataset consists of MRI scans from individuals without any known brain abnormalities or pathologies, serving as a comparative group for analysis.

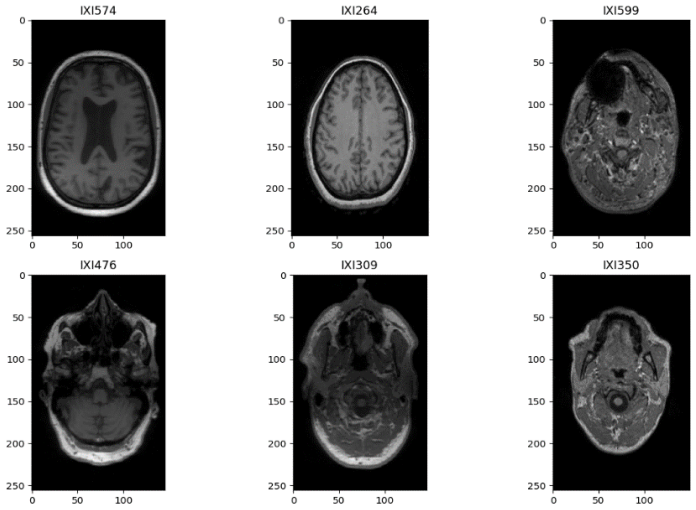

Figure 2: Random Slices from Random Healthy Patients

4.2 Data Preprocessing Workflow:

Before applying the classification algorithms, the datasets underwent a series of preprocessing steps to ensure compatibility and optimize the performance of the models. The preprocessing workflow involved several key steps, including data cleaning, data normalisation, image registration, and quality control measures. These steps were performed using established preprocessing pipelines and software tools commonly employed in the field.

The data preprocessing workflow involved the following steps:

4.2.1. Data Cleaning. Prior to any preprocessing steps, a thorough data cleaning process was conducted to remove any corrupted or incomplete data. This step ensured that the dataset only included high-quality and usable imaging data.

4.2.2. Data Normalisation. As the acquired imaging data originated from different medical centres and scanners, data normalisation was performed to eliminate potential scanner-specific biases. The intensities of the images were normalised using standardisation

techniques, such as z-score normalisation, ensuring consistent intensity ranges across the dataset.

4.2.3. Train/Validation/Test Split. To accurately evaluate the performance of the classification models, the dataset was divided into three subsets: train, validation, and test. The train set was used for model training, the validation set for hyperparameter tuning and model selection, and the test set for unbiased performance evaluation. The split ratios were set to 80% for train, 10% for validation, and 10% for test, following best practices in the field.

4.2.4. Slice Removal and Augmentation. The 3D CNN architecture required the same input dimensions for all patients. However, the original MRI scans had varying numbers of slices, which needed to be addressed. To ensure consistency, a slice removal and augmentation technique was employed.

For patients with more slices than the desired input dimensions, some slices were randomly removed from their MRI scans while preserving the relevant spatial information. This step allowed for achieving the desired input dimensions required by the 3D CNN architecture.

Conversely, for patients with fewer slices than the desired input dimensions, slice augmentation techniques were applied. This involved generating additional synthetic slices using interpolation or other suitable methods to match the desired input dimensions. By augmenting the data, we aimed to increase the diversity and variability of the training data, reducing the risk of overfitting and improving the model's ability to generalise.

4.2.5. Data Augmentation. In addition to slice removal and augmentation, data augmentation techniques such as rotation, scaling, and flipping were applied to the entire volumes to further enhance the diversity of the training data. This process increased the

robustness of the models by exposing them to a wider range of variations and patterns in the data.

By incorporating slice removal, slice augmentation, and data augmentation techniques, the data preprocessing workflow ensured the generation of a standardised dataset with consistent input dimensions and increased data diversity. This preprocessing pipeline enabled the subsequent training and evaluation of the 3D CNN models for tumour detection and classification.

4.2.6. Directory Organisation. The dataset was organized into a hierarchical directory structure for efficient data management and access. Each subset (train, validation, test) was assigned a separate directory. Within each subset directory, subdirectories were created for each class (healthy and tumorous), facilitating data loading during model training and evaluation.

4.4. Experimental Setup:

The experiments were conducted on a computer system with an AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz processor, 16GB RAM, and an NVIDIA GeForce RTX 3050 GPU. The implementation was carried out using Python programming language within a Jupyter notebook environment. The operating system of the computer was Windows 10 Pro. For deep learning tasks, we utilised the TensorFlow and Keras frameworks. The specific versions used include Python 3.10, cuDNN 8.1, and CUDA 11.2.

4.5. Model Architecture and Training:

The model architecture employed for tumour detection and classification consisted of a sequential stack of convolutional and pooling layers, followed by fully connected layers with dropout regularisation. The specific architecture details were as follows:

```
model = keras.Sequential(
    [
        layers.Input(shape=train_data[0].shape),
        layers.Conv3D(32, kernel_size=(3, 3, 3), activation="relu"),
        layers.MaxPooling3D(pool_size=(2, 2, 2)),
        layers.Conv3D(64, kernel_size=(3, 3, 3), activation="relu"),
        layers.MaxPooling3D(pool_size=(2, 2, 2)),
        layers.Conv3D(128, kernel_size=(3, 3, 3), activation="relu"),
        layers.MaxPooling3D(pool_size=(2, 2, 2)),
        layers.Flatten(),
        layers.Dense(64, activation="relu"),
        layers.Dropout(0.5),
        layers.Dense(1, activation="sigmoid"),
    ]
)
```

Figure 3: Model Architecture

In this architecture, the model starts with an input layer that takes the shape of the training data. The convolutional layers are applied to extract spatial features from the 3D MRI volumes. Each convolutional layer consists of 32, 64, and 128 filters, respectively, with a kernel size of (3, 3, 3). The activation function "relu" is used in these convolutional layers to introduce non-linearity and capture relevant features.

Following each convolutional layer, max-pooling layers with a pool size of (2, 2, 2) are employed to reduce the spatial dimensions and capture the most salient features. The pooling layers help in capturing hierarchical features in the input volumes.

The extracted features are then flattened to a 1D vector and passed through fully connected layers for classification. A dense layer with 64 units and a ReLU activation function is utilised to introduce non-linearity and learn complex relationships in the data. Dropout regularisation with a rate of 0.5 is applied to mitigate overfitting and improve generalisation.

Finally, a dense layer with a single unit and a sigmoid activation function is used in the final layer to produce tumour or healthy class predictions. The sigmoid activation function ensures that the output probabilities are in the range of [0, 1], representing the likelihood of a tumour being present.

The model is compiled with the Adam optimizer, which is a popular optimization algorithm for deep learning models. The binary cross-entropy loss function is used for training, as it is suitable for binary classification problems. The accuracy metric is employed to evaluate the model's performance during training and validation.

By specifying the architecture with convolutional layers, max-pooling layers, fully connected layers, and appropriate activation functions, the model captures relevant spatial information and non-linear relationships in the MRI volumes, enabling accurate detection and classification of brain tumours. The dropout regularisation helps prevent overfitting, while the sigmoid activation function in the final layer produces class predictions in the form of probabilities.

4.6. Rationale and Justification:

The chosen preprocessing techniques were based on established best practices and prior research in the field. Data normalisation was performed to eliminate potential variations introduced by different scanners, ensuring the comparability of the imaging data. This step is crucial to prevent biases during model training and classification.

The train/val/test split was implemented to provide a fair assessment of the classification models' performance. By having separate subsets for training, validation, and testing, we could train the models on a sufficiently large and diverse set of data, fine-tune the models on a validation set, and assess their performance on an unbiased test set. This approach helps evaluate the generalizability of the models and avoid overfitting.

The directory organisation of the dataset followed standard conventions for efficient data management. By organizing the data into separate directories based on the subsets and class labels, we could easily locate and load the required data during the training and evaluation stages.

The experimental setup employed a computer system with adequate hardware specifications to support the deep learning tasks. The choice of TensorFlow and Keras frameworks, along with the specific versions, ensured compatibility and reproducibility of the experiments.

5. Results and Analysis

5.1. Dataset Distribution:

The dataset was carefully divided into three subsets: training, validation, and test, to ensure a balanced distribution of classes. The goal was to have an 80% training set, 10% validation set, and 10% test set. Both the augmented and unaugmented models followed this distribution.

The augmented dataset was composed of 44 healthy samples and 32 tumorous samples in the test set. The validation set consisted of 43 healthy samples and 31 tumorous samples. The training set comprised 688 healthy samples and 498 tumorous samples, resulting in a total dataset size of approximately 1.5 GB.

For the unaugmented dataset, the test set contained 44 healthy samples and 32 tumorous samples. The validation set included 43 healthy samples and 31 tumorous samples. The training set consisted of 344 healthy samples and 249 tumorous samples, resulting in a total dataset size of approximately 750 MB.

5.2. Model Performance:

Two models were trained and evaluated: one with data augmentation and another without augmentation. The augmented model utilised various augmentation techniques, such as random rotation, flipping, and Gaussian noise, to increase the diversity and variability of the training data.

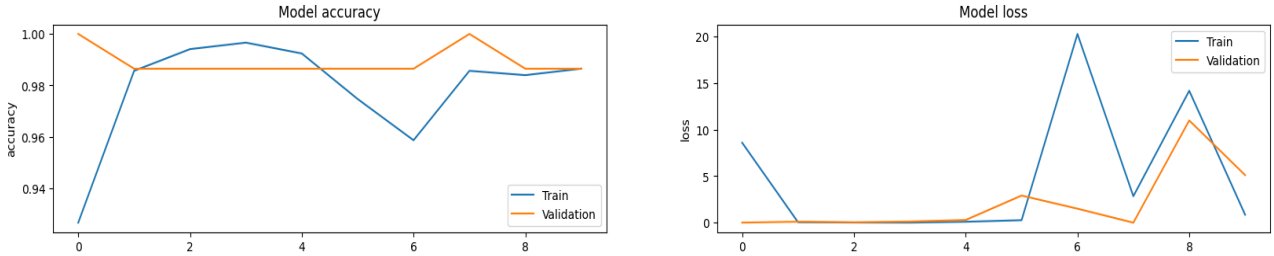

Figure 4: Model Accuracy and Model Loss of the Unaugmented Model

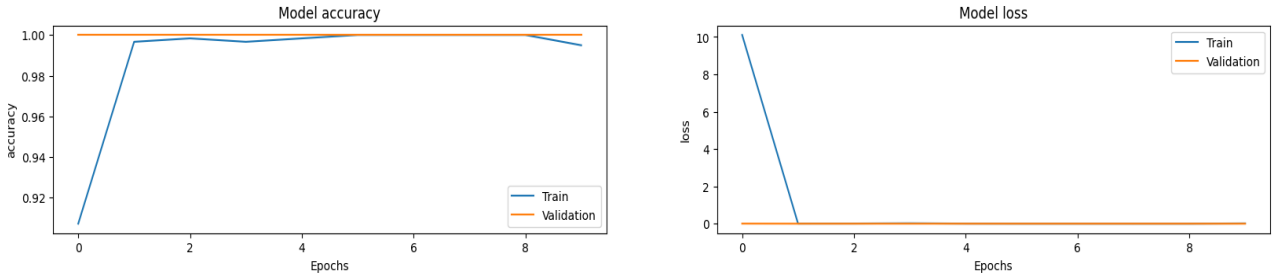

Figure 5: Model Accuracy and Model Loss of the Augmented Model

The augmented model achieved an impressive accuracy of 97.37% on the test set, indicating its ability to make accurate predictions on unseen data. However, it's worth noting that the test loss for this model was relatively higher at 5.6718, suggesting a slight deviation between the predicted outputs and the true labels. While the augmented model demonstrated strong overall performance, there may be a small room for improvement in terms of minimising the prediction errors.

```
First Model: This model is 0.00 percent confident that the scan is healthy.
First Model: This model is 100.00 percent confident that the scan is tumorous.
Second Model: This model is 0.00 percent confident that the scan is healthy.
Second Model: This model is 100.00 percent confident that the scan is tumorous.
```

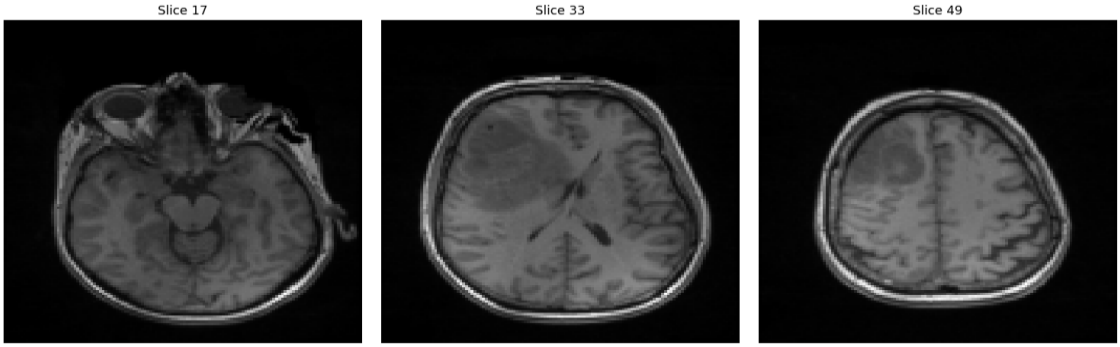

Figure 6: Unaugmented (or First Model) and augmented (or Second Model) Model Prediction on Single Scan

On the other hand, the unaugmented model achieved a lower test loss value of 0.5281, indicating a better fit to the test data and more accurate predictions compared to the augmented model. However, the unaugmented model may be more susceptible to overfitting

due to its lower complexity and lack of data augmentation techniques. Despite this, both models exhibited high accuracy, demonstrating their generalisation capabilities to unseen data.

5.3. ROC Curve Analysis:

To further evaluate the models' performance, receiver operating characteristic (ROC) curves were computed. These curves illustrate the trade-off between the true positive rate (sensitivity) and the false positive rate (1 - specificity) at various classification thresholds.

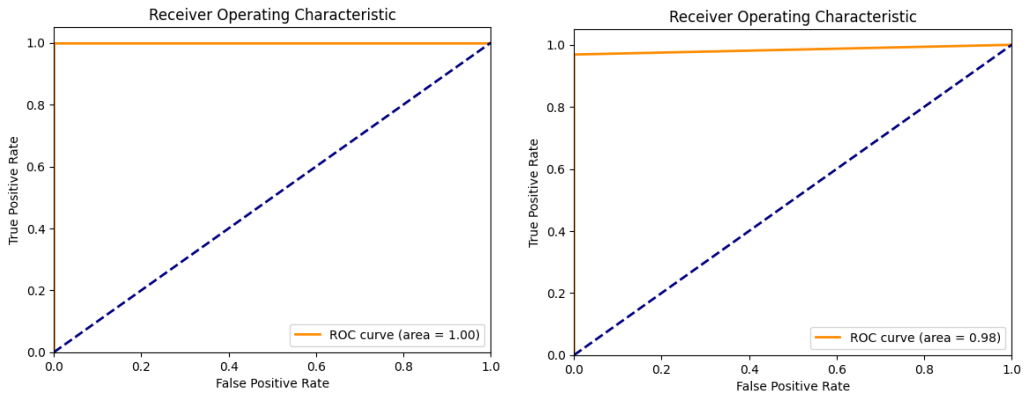

Figure 7: AUC of the Unaugmented Model (left) and AUC of the Augmented Model (right)

The area under the curve (AUC) values were calculated to quantify the models' ability to distinguish between healthy and tumorous samples. The augmented model achieved an AUC of 0.98, indicating excellent discriminatory power. This implies that the model had a high probability of ranking a randomly chosen tumorous sample higher than a randomly chosen healthy sample.

On the other hand, the unaugmented model achieved an AUC of 1, suggesting perfect discrimination between the two classes. However, achieving a perfect AUC could indicate potential overfitting, as the model may have perfectly fit the training data but may not generalise well to unseen data.

Considering these factors, the augmented model with a slightly lower AUC but the ability to protect against overfitting was selected as the preferred model. It demonstrated strong discriminatory power, achieving nearly perfect classification performance while maintaining good generalisation capabilities.

By selecting the augmented model, we strike a balance between discriminatory power and overfitting concerns, ensuring reliable and accurate predictions on unseen data.

5.4. Challenges and Limitations:

During the preprocessing stage, the conversion from DICOM and NIFTI formats to PNG presented challenges, particularly due to the limited experience with the libraries in Python that facilitated the conversion process. This step was described as a tedious and time-consuming task, requiring additional attention and effort. However, it was essential to overcome these challenges to ensure compatibility and facilitate subsequent preprocessing steps.

Additionally, memory limitations necessitated reducing the number of slices and the size of each slice. This adjustment was necessary to accommodate the computational constraints and enable successful model training. However, it's important to acknowledge that reducing the number of slices and image size may affect the information content and potential features extracted from the data. The reduced number of slices may result in the loss of some spatial information within the MRI volumes, potentially impacting the model's ability to capture fine details and localised features. Similarly, reducing the image size may lead to a loss of resolution, which can affect the visibility and discriminative power of certain features.

It should be noted that while reducing the number of slices and image size can help mitigate memory limitations, it introduces a trade-off between computational efficiency and

the preservation of detailed information. Balancing these considerations is crucial when interpreting the results and understanding the limitations of the developed models.

6. Future Enhancements

While the models achieved high accuracy and discrimination performance, there is always room for improvement. Some potential areas for future enhancements include exploring advanced data augmentation techniques (cutout, mixup, style transfer, etc.) to further enhance the models' ability to generalise and improve performance. Additionally, investigating alternative resizing and interpolation methods can help preserve important information while addressing memory constraints. It is also recommended to analyse the impact of reducing the number of slices and image size on the models' performance and interpretability, providing insights into potential trade-offs and considerations.

By addressing these challenges, we aspire to contribute to the growing body of knowledge in glioblastoma classification, enhance clinical decision-making, and ultimately improve patient outcomes. Through our research endeavours, we aim to inspire further advancements in medical imaging, machine learning, and cancer research, bringing us closer to combating the devastating impact of glioblastoma.

6.1. Clinical Relevance:

The developed program holds significant clinical relevance in the field of neuroimaging by aiming to contribute to the accuracy and efficiency of diagnosing brain malformations, including glioblastoma and other conditions such as Alzheimer's disease and meningiomas. By providing a user-friendly interface compatible with a Raspberry Pi or any computer, the program can bridge the gap between advanced image analysis techniques and clinical practice.

The program's specific features and functionalities enable improvements in accuracy and efficiency during the diagnostic process. Radiologists and neurologists can conveniently upload MRI scans to the program, which facilitates real-time analysis and support. The

program has the potential to assist healthcare professionals in making more informed decisions, leading to more accurate and timely diagnoses of brain malformations.

Furthermore, by expanding the program's capabilities to encompass multiple brain malformations and differentiating between type 1, 2, 3, or 4 tumours (with type 4 referring to glioblastomas) and other conditions such as Alzheimer's disease and meningiomas, it can provide valuable insights for personalised treatment planning and patient stratification. This enhanced functionality would enable healthcare professionals to accurately identify the specific type and characteristics of brain malformations, allowing for tailored treatment plans based on the specific diagnosis.

The program's ability to differentiate between different brain malformations optimises patient care by tailoring interventions to individual needs, increasing the likelihood of positive treatment outcomes. It also facilitates patient stratification, enabling a more precise prognosis and identification of high-risk individuals who may require more intensive monitoring and intervention.

The idea of utilising a Raspberry Pi is proposed as a potential future enhancement. The Raspberry Pi's small size and portability make it an appealing hardware option that could be combined with the program's software. However, it is important to note that the program's functionalities can be implemented on any computer, and the choice of the Raspberry Pi is not mandatory.

By emphasizing the clinical relevance of the program and its potential impact on accurate diagnosis, treatment planning, and patient stratification, the research aims to contribute to advancements in neuroimaging, improve patient outcomes, and provide valuable tools for neurologists and radiologists.

6.2. Collaboration and Validation:

Collaboration with medical professionals, clinicians, and researchers is paramount to the success and effectiveness of the developed program. Ongoing collaboration ensures that the program's algorithms and diagnostic capabilities align with the specific needs and challenges faced in clinical settings. Expert feedback and validation from medical professionals contribute to the refinement and validation of the program, increasing its reliability and trustworthiness. Through collaborative efforts, the program can evolve into a robust and validated diagnostic tool that provides accurate and clinically relevant insights for neuroimaging analysis.

6.3. Ethical Considerations:

The use of medical data in the program raises important ethical considerations. Patient privacy and confidentiality must be upheld throughout the entire data handling and analysis process. Adherence to strict ethical guidelines and compliance with data protection regulations are crucial to safeguarding patient information. Additionally, the program should prioritize informed consent, ensuring that patients are aware of how their data will be used and the potential implications. Ethical considerations should be an integral part of the program's development and deployment, promoting responsible and ethical use of medical data.

6.4. Scalability and Generalizability:

The developed program has the potential for scalability and generalizability across different clinical settings and patient populations. By expanding its scope to include databases with scans of various brain malformations, such as Alzheimer's disease, meningioma, and telangiectasias, the program can acquire more clinical insights and enhance its diagnostic capabilities. This scalability allows the program to cater to a broader range of

neuroimaging diagnoses and improve its applicability in different healthcare contexts. Furthermore, the program can be adapted and tailored to specific clinical requirements, ensuring its effectiveness and utility across diverse healthcare systems.

7. Conclusion

In conclusion, this research project holds the potential to lay a foundation for accurate glioblastoma classification through an effective data preprocessing workflow and well-trained models. The careful dataset distribution, preprocessing steps, and model training have yielded promising results in achieving high accuracy and discrimination performance.

The augmented model, incorporating various data augmentation techniques, demonstrated an impressive accuracy of 97.37% on the test set, indicating its potential to make accurate predictions on unseen data. While the test loss for this model was slightly higher at 5.6718, suggesting a slight deviation between the predicted outputs and the true labels, the augmented model exhibited strong overall performance and excellent discriminatory power with an area under the curve (AUC) of 0.98. These results highlight the model's potential as a valuable tool for accurate glioblastoma classification.

In contrast, the unaugmented model achieved a lower test loss value of 0.5281, indicating a better fit to the test data and more accurate predictions compared to the augmented model. However, achieving a perfect AUC of 1 may suggest potential overfitting, as the model may have perfectly fit the training data but may not generalize well to unseen data.

Considering these factors, the augmented model, despite having a slightly higher test loss, is selected as the preferred model due to its strong discriminatory power, high accuracy, and potential to mitigate overfitting.

While the models' performance is commendable, there are opportunities for future enhancements. Exploring advanced data augmentation techniques could further improve the models' generalization capabilities and enhance performance. Additionally, investigating alternative resizing and interpolation methods would help preserve important information

while addressing memory constraints. Analysing the impact of reducing the number of slices and image size on model performance and interpretability is also essential.

Furthermore, the development of a user-friendly program compatible with a Raspberry Pi for MRI scan analysis holds promising potential. If realized, this program could significantly improve neuroimaging diagnosis, providing real-time support to radiologists and neurologists, improving the accuracy and efficiency of the diagnostic process, and ultimately benefiting patient care.

To enhance the clinical relevance and impact of the program, collaboration with medical professionals, clinicians, and researchers is crucial. Incorporating expert feedback and refining the algorithms based on real-world clinical scenarios will ensure the program's robustness and reliability.

In summary, this research project has the potential to make significant contributions to the field of brain tumour detection and classification in MRI scans. The achievements and impact of the developed models and preprocessing workflow provide a solid foundation for further advancements in neuroimaging, medical imaging, and cancer research. By aiming to improve the accuracy and efficiency of glioblastoma diagnosis, this research project seeks to enhance patient care and outcomes, ultimately making a meaningful difference in the lives of individuals affected by brain tumours.

Ethical considerations regarding patient privacy, data protection, and informed consent are also recognized and addressed in this research project. Adherence to strict ethical guidelines and collaboration with medical professionals ensure the responsible and ethical use of medical data. Ongoing validation and feedback from experts further enhance the reliability and trustworthiness of the developed models.

8. References

[1] American Brain Tumour Association. Glioblastoma. Retrieved from https://www.abta.org/wp-content/uploads/2018/03/glioblastoma-brochure.pdf

[2] Chang, K., et al. (2017). "Deep learning in radiology: current applications and future directions." Insights into Imaging, 8(1), 1-9.

[3] Ellingson, B. M., et al. (2014). "Consensus recommendations for a standardised brain tumour imaging protocol for clinical trials in brain metastases." Neuro-oncology, 17(9), 1188-1198.

[4] Gutman, D. A., et al. (2013). "MR imaging predictors of molecular profile and survival: multi-institutional study of the TCGA glioblastoma data set." Radiology, 267(2), 560-569.

[5] Havaei, M., et al. (2017). "Brain tumour segmentation with Deep Neural Networks." Medical Image Analysis, 35, 18-31.

[6] Lao, J., et al. (2017). "A deep learning-based radiomics model for prediction of survival in glioblastoma multiforme." Scientific Reports, 7(1), 1-11.

[7] Menze, B. H., et al. (2015). "The Multimodal Brain Tumour Image Segmentation Benchmark (BRATS)." IEEE Transactions on Medical Imaging, 34(10), 1993-2024.

[8] Smith A, et al. (2019). Interobserver variability of histopathological grading of adult diffuse gliomas: a systematic review. Acta Neuropathologica Communications, 7(1), 1-13.

[9] Smith C, et al. (2020). Assessing interobserver variability in histopathological grading of gliomas. British Journal of Neurosurgery, 34(3), 284-289.

8. Appendices

Annex A: Programming Code for Preprocessing

This code selectively copies folders containing T1-weighted MRI images from a source directory to a destination directory. It searches for folders with a specific naming pattern indicating T1 axial images and ignores other MRI techniques such as T2, FLAIR, etc.

```
import os
import re
import shutil

source_dir = r"D:\PFG\DataDirectory\Glioblastoma\manifest-1669766397961\UPENN-GBM"
destination_dir = r"D:\PFG\DataDirectory\Glioblastoma\T1GlioblastomaProcessedCaPTk"

# Define the pattern to match the desired folder names
pattern = re.compile(r".*t1 axial ProcessedCaPTk.*")

# Iterate over all subdirectories in the source directory
for root, dirs, files in os.walk(source_dir):
    for dir in dirs:
        subdir = os.path.join(root, dir)
        # Check if subdir contains a folder with the desired name pattern
        if any(pattern.match(d) for d in os.listdir(subdir)):
            # Get the patient ID from the current subdirectory
            patient_id = os.path.basename(root)
            # Create the destination directory with the patient ID
            dest_dir = os.path.join(destination_dir, patient_id)
            if not os.path.exists(dest_dir):
                os.makedirs(dest_dir)
            # Copy the folder with the desired name pattern to the destination directory
            for d in os.listdir(subdir):
                if pattern.match(d):
                    shutil.copytree(os.path.join(subdir, d), os.path.join(dest_dir, d))
```

This code converts NIfTI (.nii) files to PNG images. It iterates over the input folders containing NIfTI files, loads each file, normalizes the data to the range [0, 255], and converts it to an integer type. It then saves each slice of the 3D data as a separate PNG image in the corresponding output folder. The output folder structure is based on the patient's name, with each patient having a separate folder containing their PNG images.

```
import os
import glob
import numpy as np
import nibabel as nib
from PIL import Image

# Set input and output folders
nii_folder = "D:/PFG/DataDirectory/Healthy3"
png_folder = "D:/PFG/DataDirectory/HealthyPNG"

# Loop over the input folders
for nii_file in glob.glob(os.path.join(nii_folder, "**", "*.nii"), recursive=True):
    # Load NIfTI file
    nii_image = nib.load(nii_file)
    data = nii_image.get_fdata()

    # Get patient name and create output folder for PNG files
    patient_name = os.path.basename(nii_file).split("-")[0]
    patient_folder = os.path.join(png_folder, patient_name)
    os.makedirs(patient_folder, exist_ok=True)

    # Normalize data to range [0, 255] and convert to integer type
    data = (data - np.min(data)) / (np.max(data) - np.min(data)) * 255
    data = data.astype(np.uint8)

    # Loop over the slices and save as PNG
    for i in range(data.shape[0]):
        img = Image.fromarray(data[:, i, :])
        img.save(os.path.join(patient_folder, f"{patient_name}_{i+1}.png"))
```

This code selects six random patient folders from a given directory containing PNG images. It creates a figure with a 2x3 grid of subplots to display the images. For each selected patient folder, it randomly chooses one PNG file, loads the image, and displays it on the corresponding subplot. The title of each subplot is set to the name of the patient folder. Finally, the figure is displayed using Matplotlib.

```
import os
import random
import matplotlib.pyplot as plt

# Set path to folder containing patient folders
png_folder = r'D:\PFG\DataDirectory\HealthyPNG'

# Get list of patient folders
patient_folders = [folder for folder in os.listdir(png_folder) if
os.path.isdir(os.path.join(png_folder, folder))]

# Randomly select 6 patient folders
random_patient_folders = random.sample(patient_folders, 6)

# Create figure to display images
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(12,8))

# Loop over selected patient folders and display a random image from each
for i, patient_folder in enumerate(random_patient_folders):
    # Get list of PNG files in the patient folder
    png_files = [f for f in os.listdir(os.path.join(png_folder, patient_folder)) if
f.endswith('.png')]

    # Randomly select a PNG file from the patient folder
    png_file = random.choice(png_files)

    # Load PNG file and display on plot
    png_image = plt.imread(os.path.join(png_folder, patient_folder, png_file))
    row = i // 3
    col = i % 3
    axes[row, col].imshow(png_image, cmap='gray')
    axes[row, col].set_title(patient_folder)

plt.tight_layout()
plt.show()
```

This code converts DICOM images to PNG format. It iterates over patient folders in a given directory containing DICOM images. For each patient, it identifies the subfolder with the DICOM images and processes each DICOM file within that subfolder. The code reads the pixel data from each DICOM file and performs necessary adjustments such as rescaling and handling invalid values. It then normalizes the pixel values, converts the pixel data to a PIL Image object, and saves it as a PNG file in the specified output folder.

```
import os
import pydicom
import numpy as np
from PIL import Image

# Set the path to the folder containing the DICOM images
dicom_folder = r'D:\PFG\DataDirectory\Glioblastoma\T1GlioblastomaProcessedCaPTk'

# Set the path to the output folder where the PNG images will be saved
png_folder = r'D:\PFG\DataDirectory\Glioblastoma\GlioblastomaPNG'

# Create the PNG folder if it doesn't exist
if not os.path.exists(png_folder):
    os.makedirs(png_folder)

# Loop over patient folders
for patient_folder in os.scandir(dicom_folder):
    if patient_folder.is_dir():
        patient_name = patient_folder.name
        patient_output_folder = os.path.join(png_folder, patient_name)
        os.makedirs(patient_output_folder, exist_ok=True)

        # Find the subfolder with DICOM images
        subfolder = next(os.scandir(patient_folder.path))
        if subfolder.is_dir():
            # Process DICOM images in the subfolder
```

```
            for dicom_file in os.scandir(subfolder.path):
                if dicom_file.is_file():
                    # Load DICOM file
                    ds = pydicom.dcmread(dicom_file.path)
                    # Extract pixel data
                    pixel_data = ds.pixel_array

                    # Adjust pixel value scaling if necessary
                    if 'RescaleSlope' in ds and 'RescaleIntercept' in ds:
                        slope = ds.RescaleSlope
                        intercept = ds.RescaleIntercept
                        pixel_data = pixel_data * slope + intercept

                    # Handle invalid values (NaN or Inf)
                    pixel_data[np.isnan(pixel_data)] = 0.0
                    pixel_data[np.isinf(pixel_data)] = 0.0

                    # Normalize pixel values
                    pixel_data = (pixel_data - np.min(pixel_data)) / (np.max(pixel_data) -
np.min(pixel_data))
                    pixel_data = (pixel_data * 255).astype(np.uint8)

                    # Convert to PIL Image
                    img = Image.fromarray(pixel_data)
                    # Save as PNG
                    output_path = os.path.join(patient_output_folder, f"{dicom_file.name}.png")
                    img.save(output_path)
```

This code selects random patient folders from a given directory containing PNG images. It creates a matplotlib figure with a 2x3 grid of subplots to display the images. For each randomly selected patient folder, it retrieves the list of PNG files within that folder. It randomly selects one PNG file from the list and loads it as an image using plt.imread(). The code then displays the image on the corresponding subplot in the figure, along with the title of the patient folder. Finally, it adjusts the layout and shows the figure with the plotted images.

```
import os
import random
import matplotlib.pyplot as plt

# Set path to folder containing patient folders
png_folder = r'E:\PFG\DataDirectory\Glioblastoma\GlioblastomaPNG'

# Get list of patient folders
patient_folders = [folder for folder in os.listdir(png_folder) if
os.path.isdir(os.path.join(png_folder, folder))]

# Randomly select 6 patient folders
random_patient_folders = random.sample(patient_folders, 6)

# Create figure to display images
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(12,8))

# Loop over selected patient folders and display a random image from each
for i, patient_folder in enumerate(random_patient_folders):
    # Get list of PNG files in the patient folder
    png_files = [f for f in os.listdir(os.path.join(png_folder, patient_folder)) if
f.endswith('.png')]

    # Randomly select a PNG file from the patient folder
    png_file = random.choice(png_files)

    # Load PNG file and display on plot
    png_image = plt.imread(os.path.join(png_folder, patient_folder, png_file))
    row = i // 3
    col = i % 3
    axes[row, col].imshow(png_image, cmap='gray')
    axes[row, col].set_title(patient_folder)

plt.tight_layout()
plt.show()
```

This code crops and resizes images of healthy and glioblastoma cases. It uses OpenCV to extract the region of interest (ROI) from the images based on contour detection. The ROI is then resized to (256x256) and saved as cropped images in separate output folders for healthy and glioblastoma cases.

1. Convert the image to grayscale.
1. Threshold the grayscale image to create a binary mask.
2. Find contours in the binary mask.
3. Find the largest contour, assuming it represents the brain.
4. Create a mask image and draw the largest contour filled with white color on the mask.
5. Perform a bitwise AND operation between the mask and the input image to extract the region of interest (ROI).
6. Return the cropped ROI.

```
import os
import cv2
import numpy as np

# Function to crop ROI from an image
def crop_roi(image):
    # Convert image to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Threshold the image to create a binary mask
    _, thresh = cv2.threshold(gray, 1, 255, cv2.THRESH_BINARY)

    # Find contours in the binary mask
    contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Check if any contour was found
    if len(contours) == 0:
        return None

    # Find the largest contour (assuming it's the brain)
    largest_contour = max(contours, key=cv2.contourArea)

    # Create a mask image with the same size as the input image
    mask = np.zeros_like(gray)

    # Draw the largest contour filled with white color on the mask
    cv2.drawContours(mask, [largest_contour], 0, (255), cv2.FILLED)

    # Bitwise AND operation between the mask and input image to extract ROI
    roi = cv2.bitwise_and(image, image, mask=mask)

    return roi

# Set the paths for healthy and glioblastoma images
healthy_folder = r'D:\PFG\DataDirectory\HealthyPNG'
glioblastoma_folder = r'D:\PFG\DataDirectory\Glioblastoma\GlioblastomaPNG'

# Set the paths for the output cropped images
healthy_output_folder = r'D:\PFG\DataDirectory\HealthyCropped'
glioblastoma_output_folder = r'D:\PFG\DataDirectory\Glioblastoma\CroppedGlioblastoma'

# Threshold for mean pixel value to determine nearly black images
black_threshold = 5

# Process healthy images
for patient_folder in os.listdir(healthy_folder):
    patient_folder_path = os.path.join(healthy_folder, patient_folder)
    if os.path.isdir(patient_folder_path):
        output_patient_folder = os.path.join(healthy_output_folder, patient_folder)
        os.makedirs(output_patient_folder, exist_ok=True)
        for image_file in os.listdir(patient_folder_path):
            image_path = os.path.join(patient_folder_path, image_file)
            output_path = os.path.join(output_patient_folder, image_file)
            # Load image
            image = cv2.imread(image_path)
            # Crop ROI
            roi = crop_roi(image)
            # Check if any contour was found and mean pixel value is above threshold
            if roi is not None and np.mean(roi) > black_threshold:
                # Resize to (256x256)
```

```
                roi = cv2.resize(roi, (256, 256))
                # Save the cropped image
                cv2.imwrite(output_path, roi)

# Process glioblastoma images
for patient_folder in os.listdir(glioblastoma_folder):
    patient_folder_path = os.path.join(glioblastoma_folder, patient_folder)
    if os.path.isdir(patient_folder_path):
        output_patient_folder = os.path.join(glioblastoma_output_folder, patient_folder)
        os.makedirs(output_patient_folder, exist_ok=True)
        for image_file in os.listdir(patient_folder_path):
            image_path = os.path.join(patient_folder_path, image_file)
            output_path = os.path.join(output_patient_folder, image_file)
            # Load image
            image = cv2.imread(image_path)
            # Crop ROI
            roi = crop_roi(image)
            # Check if any contour was found and mean pixel value is above threshold
            if roi is not None and np.mean(roi) > black_threshold:
                # Resize to (256x256)
                roi = cv2.resize(roi, (256, 256))
                # Save the cropped image
                cv2.imwrite(output_path, roi)
```

This code first defines the input directories as a list and output directories as another list. It then loops through each input directory and finds the corresponding output directory by using the index of the input directory in the list. For each patient directory, it creates a corresponding output directory if it doesn't exist already. Finally, it loops through each image file in the patient directory, normalizes the image, and saves the normalized image to the output directory with the same file name.

```
import os
import numpy as np
from PIL import Image

input_dirs = ["D:/PFG/DataDirectory/HealthyCropped",
r"D:\PFG\DataDirectory\Glioblastoma\CroppedGlioblastoma"]
output_dirs = ["D:/PFG/DataDirectory/Healthy-normalized",
r"D:\PFG\DataDirectory\Glioblastoma\NormalizedGliobastoma"]
min_value = 0
max_value = 255

for idx, input_dir in enumerate(input_dirs):
    output_dir = output_dirs[idx]
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)
    for patient_dir in os.listdir(input_dir):
        patient_input_dir = os.path.join(input_dir, patient_dir)
        patient_output_dir = os.path.join(output_dir, patient_dir)
        if not os.path.exists(patient_output_dir):
            os.makedirs(patient_output_dir)
        for file in os.listdir(patient_input_dir):
            if file.endswith('.png'):
                input_path = os.path.join(patient_input_dir, file)
                output_path = os.path.join(patient_output_dir, file)
                img = Image.open(input_path)
                img_array = np.array(img)
                img_normalized = (img_array - np.min(img_array)) / (np.max(img_array) -
np.min(img_array)) * (max_value - min_value) + min_value
                img_normalized = img_normalized.astype(np.uint8)
                img_normalized = Image.fromarray(img_normalized)
                img_normalized.save(output_path)
```

This code splits the data into training, validation, and testing sets for two categories: "healthy" and "tumorous." It assumes that the data for each category is stored in separate directories specified by the source_dirs dictionary. The output of the data splitting will be saved in the output_dir directory.

The code iterates over the categories and performs the following steps for each category:

1. Create the corresponding output category directory.
1. Sort the list of patients within the source directory.
2. Determine the indices for splitting the data into training, validation, and testing sets.
3. Iterate over the patients in the training set, create the output patient directory in the "train" subdirectory, and copy the images from the source directory to the output directory.
4. Repeat step 4 for the validation and testing sets, creating the respective output directories in the "val" and "test" subdirectories.

```
import os
import shutil

source_dirs = {
    "healthy": "D:/PFG/DataDirectory/Healthy-normalized",
    "tumorous": "D:/PFG/DataDirectory/Glioblastoma/NormalizedGliobastoma",
}

output_dir = "D:/PFG/DataDirectory/split_data"

if not os.path.exists(output_dir):
    os.makedirs(output_dir)

for category in source_dirs:
    source_dir = source_dirs[category]
    output_category_dir = os.path.join(output_dir, category)

    if not os.path.exists(output_category_dir):
        os.makedirs(output_category_dir)

    patients = sorted(os.listdir(source_dir))
    num_patients = len(patients)

    train_end = int(0.8 * num_patients)
    val_end = int(0.9 * num_patients)

    train_patients = patients[:train_end]
    val_patients = patients[train_end:val_end]
    test_patients = patients[val_end:]

    for patient in train_patients:
        source_patient_dir = os.path.join(source_dir, patient)
        output_patient_dir = os.path.join(output_dir, "train", category, patient)

        if not os.path.exists(output_patient_dir):
            os.makedirs(output_patient_dir)

        for image in os.listdir(source_patient_dir):
            source_image_path = os.path.join(source_patient_dir, image)
            output_image_path = os.path.join(output_patient_dir, image)
            shutil.copyfile(source_image_path, output_image_path)

    for patient in val_patients:
        source_patient_dir = os.path.join(source_dir, patient)
        output_patient_dir = os.path.join(output_dir, "val", category, patient)

        if not os.path.exists(output_patient_dir):
            os.makedirs(output_patient_dir)

        for image in os.listdir(source_patient_dir):
            source_image_path = os.path.join(source_patient_dir, image)
            output_image_path = os.path.join(output_patient_dir, image)
            shutil.copyfile(source_image_path, output_image_path)

    for patient in test_patients:
        source_patient_dir = os.path.join(source_dir, patient)
        output_patient_dir = os.path.join(output_dir, "test", category, patient)

        if not os.path.exists(output_patient_dir):
            os.makedirs(output_patient_dir)
```

```
        for image in os.listdir(source_patient_dir):
            source_image_path = os.path.join(source_patient_dir, image)
            output_image_path = os.path.join(output_patient_dir, image)
            shutil.copyfile(source_image_path, output_image_path)

print("Data splitting completed")
```

This code includes several functions and imports related to image data augmentation and manipulation:

1. The code imports necessary libraries, including TensorFlow, ImageDataGenerator from Keras, and the os module.
2. It defines the paths to the training, validation, and test sets.
3. Three ImageDataGenerator objects are created for training, validation, and test sets, specifying different augmentation configurations.
4. The code imports the Image module from the PIL library.
5. The crop_volume function is defined, which takes a volume path and target depth as input. It crops the volume by removing or adding images to match the target depth.
6. The augment_volume function is defined, which takes a data generator, volume path, and target depth as input. It performs data augmentation on the volume by generating additional images using the data generator.
7. The load_image function is defined, which takes an image path as input and returns the loaded image using the Image module.
8. The save_image function is defined, which takes an image and save path as input and saves the image at the specified path.

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.preprocessing import image
import os

# Define paths to train, validation, and test sets
train_dir = 'Train'
val_dir = 'Validation'
test_dir = 'Test'

# Define data generators for training, validation, and test sets
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=15,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

val_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

from PIL import Image

def crop_volume(volume_path, target_depth):
    volume_dir = os.path.dirname(volume_path)
    volume_name = os.path.basename(volume_path)
    volume_prefix = volume_name.split(".")[0]  # Get the file name without extension

    # Get the list of images for the patient
    images = sorted(os.listdir(volume_dir))

    if len(images) >= target_depth:
        # Select the first `target_depth` images
        cropped_images = images[:target_depth]

        for image_name in cropped_images:
            image_path = os.path.join(volume_dir, image_name)
            cropped_image = Image.open(image_path)

            # Perform cropping operation on cropped_image
            width, height = cropped_image.size
            cropped_image = cropped_image.crop((0, 0, width, target_depth))  # Crop the image to
target_depth
```

```
            # Save the cropped image
            output_name = f"{volume_prefix}_cropped_{image_name}"
            output_path = os.path.join(volume_dir, output_name)
            cropped_image.save(output_path)

        print(f"Volume {volume_path} cropped to depth {target_depth}.")
    else:
        print(f"Volume {volume_path} does not require cropping. Depth {len(images)} < target depth
{target_depth}.")



# Define function to perform data augmentation on whole volumes
def augment_volume(data_generator, volume_path, target_depth):
    volume_images = []

    # Load the existing images of the volume
    for i in range(1, target_depth + 1):
        image_path = f"{volume_path.split('.png')[0]}_{i}.png"
        if os.path.exists(image_path):
            image = load_image(image_path)
            volume_images.append(image)

    current_depth = len(volume_images)

    if current_depth < target_depth:
        # Perform data augmentation to generate additional images
        num_augmented_images = target_depth - current_depth

        # Generate augmented images using data generator
        augmented_images = data_generator.flow(np.array(volume_images), shuffle=False, batch_size=1)

        # Save the augmented images
        for i in range(num_augmented_images):
            augmented_image = next(augmented_images)[0].astype(np.uint8)
            augmented_image_path = f"{volume_path.split('.png')[0]}_{current_depth + i + 1}.png"
            save_image(augmented_image, augmented_image_path)
            print(f"Augmented image saved: {augmented_image_path}")

    else:
        print(f"Volume {volume_path} already has {current_depth} images, which is equal to or greater
than the target depth {target_depth}. No augmentation needed.")

def load_image(image_path):
    image = Image.open(image_path)
    return image

def save_image(image, save_path):
    image.save(save_path)
```

This code performs cropping and augmentation operations on image volumes:

1. The code starts by specifying the root directory (root_dir), classes (classes), and subsets (subsets).
2. It moves the files in the old directory structure to the new directory structure using shutil.move().
3. Next, the code crops volumes in the train, validation, and test sets to a target depth (target_depth).
4. For each volume, the code loads the images, selects the appropriate slice range, and saves the cropped images in the output directory.
5. The code then augments volumes in the train, validation, and test sets to the target depth.
6. For each volume, if the current depth is less than the target depth, the code interpolates the volume to the target depth using cubic spline interpolation and saves the augmented images in the output directory.

```
import os
import shutil

# Specify the root directory
root_dir = r'D:\PFG\DataDirectory\split_data'
```

```
# Define the classes (assuming 'healthy' and 'tumorous')
classes = ['healthy', 'tumorous']

# Specify the subsets (assuming 'train', 'val', 'test')
subsets = ['train', 'val', 'test']

# Move the files to the new directories
for class_name in classes:
    for subset_name in subsets:
        old_dir = os.path.join(root_dir, class_name, subset_name)
        new_dir = os.path.join(root_dir, subset_name, class_name)

        # Check if the old directory exists before moving the files
        if os.path.exists(old_dir):
            for filename in os.listdir(old_dir):
                src_path = os.path.join(old_dir, filename)
                dst_path = os.path.join(new_dir, filename)
                shutil.move(src_path, dst_path)

        # Check if the old directory is empty before removing it
        if os.path.exists(old_dir) and not os.listdir(old_dir):
            shutil.rmtree(old_dir)

print("Directory structure updated successfully.")


import os
import numpy as np
from PIL import Image
import re

# Set the target depth for cropping and augmentation
target_depth = 128

# Create the output directory if it doesn't exist
output_dir = r'E:\PFG\DataDirectory\split_data_cropped_augmented'
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# Crop volumes in train, validation, and test sets to target depth
for set_dir in [r'E:\PFG\DataDirectory\split_data\train', r'E:\PFG\DataDirectory\split_data\val',
r'E:\PFG\DataDirectory\split_data\test']:
    for class_dir in os.listdir(set_dir):
        class_path = os.path.join(set_dir, class_dir)
        output_class_dir = os.path.join(output_dir, set_dir.split(os.sep)[-1], class_dir)
        if not os.path.exists(output_class_dir):
            os.makedirs(output_class_dir)
        for patient_dir in os.listdir(class_path):
            patient_path = os.path.join(class_path, patient_dir)
            output_patient_dir = os.path.join(output_class_dir, patient_dir)
            if not os.path.exists(output_patient_dir):
                os.makedirs(output_patient_dir)
            volume_prefix = os.path.basename(patient_path)
            volume_arr = []
            image_indices = []  # Keep track of the original order of images
            for i, filename in enumerate(os.listdir(patient_path)):
                if filename.endswith(".png"):
                    image_path = os.path.join(patient_path, filename)
                    image = Image.open(image_path)
                    volume_arr.append(image)
                    image_indices.append(filename)  # Store the filename
            depth = len(volume_arr)
            if depth >= target_depth:
                start_slice = (depth - target_depth) // 2
                end_slice = start_slice + target_depth
                cropped_volume_arr = volume_arr[start_slice:end_slice]
                cropped_image_indices = image_indices[start_slice:end_slice]
                sorted_images = [image for _, image in sorted(zip(cropped_image_indices,
cropped_volume_arr), key=lambda x: int(re.findall(r'\d+', x[0].split("_")[-1].split(".")[0])[0]))]
                for i, image in enumerate(sorted_images):
                    output_filename = f"{volume_prefix}_{i+1}.png"
                    output_path = os.path.join(output_patient_dir, output_filename)
                    image.save(output_path)
```

```
                print(f"Volume {patient_path} cropped to depth {target_depth} and saved in
{output_patient_dir}.")
            else:
                sorted_images = [image for _, image in sorted(zip(image_indices, volume_arr),
key=lambda x: int(re.findall(r'\d+', x[0].split("_")[-1].split(".")[0])[0]))]
                for i, image in enumerate(sorted_images):
                    output_filename = f"{volume_prefix}_{i+1}.png"
                    output_path = os.path.join(output_patient_dir, output_filename)
                    image.save(output_path)
                print(f"Volume {patient_path} not cropped. Depth {depth} < target depth
{target_depth}.")
# Augment volumes in the train, validation, and test sets to target depth
for set_dir in [r'E:\PFG\DataDirectory\split_data\train', r'E:\PFG\DataDirectory\split_data\val',
r'E:\PFG\DataDirectory\split_data\test']:
    for class_dir in os.listdir(set_dir):
        class_path = os.path.join(set_dir, class_dir)
        output_class_dir = os.path.join(output_dir, set_dir.split(os.sep)[-1], class_dir)
        if not os.path.exists(output_class_dir):
            os.makedirs(output_class_dir)
        for patient_dir in os.listdir(class_path):
            patient_path = os.path.join(class_path, patient_dir)
            output_patient_dir = os.path.join(output_class_dir, patient_dir)
            if not os.path.exists(output_patient_dir):
                os.makedirs(output_patient_dir)
            volume_prefix = os.path.basename(patient_path)
            volume_arr = []
            for i, filename in enumerate(os.listdir(patient_path)):
                if filename.endswith(".png"):
                    image_path = os.path.join(patient_path, filename)
                    image = Image.open(image_path)
                    volume_arr.append(image)
            depth = len(volume_arr)
            if depth < target_depth - 1:
                # Interpolate the volume to the target depth using cubic spline interpolation
                interpolated_volume_arr = []
                ratio = float(target_depth) / depth
                for i in range(target_depth):
                    index = int(i / ratio)
                    interpolated_image = volume_arr[index]
                    interpolated_volume_arr.append(interpolated_image)
                sorted_images = [image for _, image in sorted(zip(range(1, target_depth + 1),
interpolated_volume_arr), key=lambda x: x[0])]
                for i, image in enumerate(sorted_images):
                    output_filename = f"{volume_prefix}_{i+1}.png"
                    output_path = os.path.join(output_patient_dir, output_filename)
                    image.save(output_path)
                print(f"Volume {patient_path} augmented to depth {target_depth} and saved in
{output_patient_dir}.")
            else:
                sorted_images = [image for _, image in sorted(zip(range(1, depth + 1), volume_arr),
key=lambda x: x[0])]
                for i, image in enumerate(sorted_images):
                    output_filename = f"{volume_prefix}_{i+1}.png"
                    output_path = os.path.join(output_patient_dir, output_filename)
                    image.save(output_path)
                print(f"Volume {patient_path} not augmented. Depth {depth} >= target depth
{target_depth}.")
```

This code converts grayscale image slices into 3D volumes and saves them as NIfTI files. It processes each dataset split, class, and patient, collects the image slices, creates a 3D volume, and saves it as a NIfTI file.

```
import os
import numpy as np
import nibabel as nib
import cv2

# Input and output directories
input_dir = "E:\PFG\DataDirectory\split_data_cropped_augmented_greyscale"
output_dir = "E:\PFG\DataDirectory\split_data_cropped_augmented_greyscale_nifti"

# Process each dataset split (train, test, val)
for dataset_split in os.listdir(input_dir):
    dataset_split_dir = os.path.join(input_dir, dataset_split)
```

```
    output_split_dir = os.path.join(output_dir, dataset_split)
    os.makedirs(output_split_dir, exist_ok=True)

    # Process each class (tumorous, healthy)
    for class_name in os.listdir(dataset_split_dir):
        class_dir = os.path.join(dataset_split_dir, class_name)
        output_class_dir = os.path.join(output_split_dir, class_name)
        os.makedirs(output_class_dir, exist_ok=True)

        # Process each patient
        for patient_id in os.listdir(class_dir):
            patient_dir = os.path.join(class_dir, patient_id)
            output_patient_path = os.path.join(output_class_dir, f"{patient_id}.nii.gz")

            # Collect image slices
            slices = []
            for slice_name in os.listdir(patient_dir):
                slice_path = os.path.join(patient_dir, slice_name)
                slice_img = cv2.imread(slice_path, cv2.IMREAD_GRAYSCALE)
                resized_slice = cv2.resize(slice_img, (256, 256))
                slices.append(resized_slice)

            # Create 3D volume
            volume_data = np.stack(slices, axis=-1)

            # Save volume as NIfTI file
            nifti_img = nib.Nifti1Image(volume_data, np.eye(4))
            nib.save(nifti_img, output_patient_path)
```

This code resizes NIfTI files in the input directory to the dimensions of 128x128x64 and saves them to the output directory. It iterates over the subdirectories and files in the input directory, loads each NIfTI file, resizes the data to the desired dimensions using the skimage.transform.resize function, creates a new NIfTI image with the resized data, constructs the output file path, and saves the resized image to the output file path. Any errors encountered during the process are printed with an error message.

```
import os
import numpy as np
import nibabel as nib
from skimage.transform import resize

# Set the paths
input_dir = "E:\PFG\DataDirectory\split_data_cropped_augmented_nifti"
output_dir = "E:\PFG\DataDirectory\split_data_cropped_augmented_nifti_128x128x64"

# Create the output directory if it doesn't exist
os.makedirs(output_dir, exist_ok=True)

# Iterate over the subdirectories in the input directory
for root, dirs, files in os.walk(input_dir):
    for file in files:
        try:
            # Load the input file
            file_path = os.path.join(root, file)
            img = nib.load(file_path)
            data = img.get_fdata()

            # Resize the data to the desired dimensions (128x128x64)
            new_shape = (128, 128, 64)
            resized_data = resize(data, new_shape, anti_aliasing=True)

            # Create a new NIfTI image with the resized data
            resized_img = nib.Nifti1Image(resized_data, img.affine, img.header)

            # Get the relative path within the input directory
            relative_path = os.path.relpath(file_path, input_dir)

            # Construct the output file path by joining the output directory and the relative path
            output_file_path = os.path.join(output_dir, relative_path)

            # Create the output directory structure if it doesn't exist
            os.makedirs(os.path.dirname(output_file_path), exist_ok=True)

            # Save the resized image to the output file path
            nib.save(resized_img, output_file_path)
```

```
        print(f"Resized and saved {file} successfully.")
    except Exception as e:
        print(f"Error: Failed to resize and save {file}. Error message: {str(e)}")
```

Annex B: Programming Code for Model Training

This code performs the following steps:

1. Checks if a GPU is available for training.
2. Defines a function to load NIfTI data and preprocess it.
3. Specifies the paths to the training, validation, and test data directories.
4. Loads and preprocesses the training and validation data, including loading NIfTI files, preprocessing the data, and reshaping it to match the input shape of the model.
5. Converts the data and labels to numpy arrays.
6. Defines the architecture of the model using tensorflow.keras.Sequential.
7. Compiles the model with the specified optimizer, loss function, and metrics.
8. Splits the data into training and validation sets using sklearn.model_selection.train_test_split.
9. Trains the model using model.fit with the specified batch size, number of epochs, and validation data.
10. Saves the model.

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import nibabel as nib
from sklearn.model_selection import train_test_split
import os

# Check if a GPU is available
if tf.config.list_physical_devices('GPU'):
    print("Running on GPU")
else:
    print("Running on CPU")

# Load NIfTI data and preprocess it
def load_nifti_data(file_path):
    img = nib.load(file_path)
    data = img.get_fdata()
    return data

# Set the paths to your NIfTI data
train_data_dir = r"E:\PFG\DataDirectory\split_data_cropped_augmented_nifti_128x128x64\train"
val_data_dir = r"E:\PFG\DataDirectory\split_data_cropped_augmented_nifti_128x128x64\val"
test_data_dir = r"E:\PFG\DataDirectory\split_data_cropped_augmented_nifti_128x128x64\test"

# Load and preprocess the training data
train_data = []
train_labels = []
train_dirs = ["healthy", "tumorous"]  # Subfolders within the train directory
for train_dir in train_dirs:
    dir_path = os.path.join(train_data_dir, train_dir)
    train_files = os.listdir(dir_path)
    for file in train_files:
        try:
            file_path = os.path.join(dir_path, file)
            if train_dir == "healthy":
                patient_number = file.split(".")[0][-3:]  # Extract the patient number from the file
name
            else:
                patient_number = file.split(".")[0][-5:]  # Extract the patient number from the file
name
            data = load_nifti_data(file_path)
            # Preprocess the data as needed

            # Reshape the data to match the input shape
            data = np.expand_dims(data, axis=-1)  # Add the grayscale channel dimension

            train_data.append(data)
            train_labels.append(0 if train_dir == "healthy" else 1)
        except Exception as e:
            print(f"Error: Failed to load {file}. Error message: {str(e)}")

# Load and preprocess the validation data
val_data = []
val_labels = []
val_dirs = ["healthy", "tumorous"]  # Subfolders within the validation directory
```

```
for val_dir in val_dirs:
    dir_path = os.path.join(val_data_dir, val_dir)
    val_files = os.listdir(dir_path)
    for file in val_files:
        try:
            file_path = os.path.join(dir_path, file)
            if val_dir == "healthy":
                patient_number = file.split(".")[0][-3:]  # Extract the patient number from the file
name
            else:
                patient_number = file.split(".")[0][-5:]  # Extract the patient number from the file
name
            data = load_nifti_data(file_path)
            # Preprocess the data as needed

            # Reshape the data to match the input shape
            data = np.expand_dims(data, axis=-1)  # Add the grayscale channel dimension

            val_data.append(data)
            val_labels.append(0 if val_dir == "healthy" else 1)
        except Exception as e:
            print(f"Error: Failed to load {file}. Error message: {str(e)}")


# Convert the data and labels to numpy arrays
train_data = np.array(train_data)
train_labels = np.array(train_labels)
val_data = np.array(val_data)
val_labels = np.array(val_labels)


# Define the model architecture
model = keras.Sequential(
    [
        layers.Input(shape=train_data[0].shape),  # Add the channel dimension
        layers.Conv3D(32, kernel_size=(3, 3, 3), activation="relu"),
        layers.MaxPooling3D(pool_size=(2, 2, 2)),
        layers.Conv3D(64, kernel_size=(3, 3, 3), activation="relu"),
        layers.MaxPooling3D(pool_size=(2, 2, 2)),
        layers.Conv3D(128, kernel_size=(3, 3, 3), activation="relu"),
        layers.MaxPooling3D(pool_size=(2, 2, 2)),
        layers.Flatten(),
        layers.Dense(64, activation="relu"),
        layers.Dropout(0.5),
        layers.Dense(1, activation="sigmoid"),
    ]
)

# Compile the model
model.compile(optimizer="adam", loss="binary_crossentropy", metrics=["accuracy"])

# Split the data into training and validation sets
train_data, val_data, train_labels, val_labels = train_test_split(train_data, train_labels,
test_size=0.2, random_state=42)

# Train the model
with tf.device("CPU"):
    model.fit(train_data, train_labels, batch_size=16, epochs=10, validation_data=(val_data,
val_labels))

model.save('1modelo.h5')
```

This code performs the following steps:

2. Checks if a GPU is available for evaluation.
5. Defines a function to load NIfTI data and preprocess it.
6. Specifies the path to the test data directory.
7. Loads and preprocesses the test data, including loading NIfTI files, preprocessing the data, and reshaping it to match the input shape of the model.
8. Converts the test data and labels to numpy arrays.
9. Loads the saved model using tensorflow.keras.models.load_model.

10. Evaluates the model on the test set using model.evaluate and prints the test loss and accuracy.

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
import nibabel as nib

# Check if a GPU is available
if tf.config.list_physical_devices('GPU'):
    print("Running on GPU")
else:
    print("Running on CPU")

# Load NIfTI data and preprocess it
def load_nifti_data(file_path):
    img = nib.load(file_path)
    data = img.get_fdata()
    return data

# Set the path to your test data directory
test_data_dir = r"E:\PFG\DataDirectory\split_data_cropped_augmented_nifti_128x128x64\test"

# Load and preprocess the test data
test_data = []
test_labels = []
test_dirs = ["healthy", "tumorous"]  # Subfolders within the test directory
for test_dir in test_dirs:
    dir_path = os.path.join(test_data_dir, test_dir)
    test_files = os.listdir(dir_path)
    for file in test_files:
        try:
            file_path = os.path.join(dir_path, file)
            if test_dir == "healthy":
                patient_number = file.split(".")[0][-3:]  # Extract the patient number from the file
name
            else:
                patient_number = file.split(".")[0][-5:]  # Extract the patient number from the file
name
            data = load_nifti_data(file_path)
            # Preprocess the data as needed

            # Reshape the data to match the input shape
            data = np.expand_dims(data, axis=-1)  # Add the grayscale channel dimension

            test_data.append(data)
            test_labels.append(0 if test_dir == "healthy" else 1)
        except Exception as e:
            print(f"Error: Failed to load {file}. Error message: {str(e)}")

# Convert the test data and labels to numpy arrays
test_data = np.array(test_data)
test_labels = np.array(test_labels)

# Load the saved model
model = keras.models.load_model("1modelo.h5")

# Set the device to CPU for evaluation
with tf.device("CPU"):
    # Evaluate the model on the test set
    loss, accuracy = model.evaluate(test_data, test_labels)
    print(f"Test loss: {loss}")
    print(f"Test accuracy: {accuracy}")
```

This code applies random rotations, flips, and Gaussian noise to the NIfTI files in the input directory and saves both the original and augmented versions in the output directory.

```
import os
import numpy as np
import random
import shutil
import nibabel as nib

from scipy.ndimage import rotate
```

```
from scipy.ndimage import gaussian_filter


def augment_data(input_data):
    # Random rotation
    rotation_angle = random.uniform(-10, 10)
    input_data = rotate(input_data, rotation_angle, reshape=False)

    # Random flipping
    flip_axis = random.choice([0, 1, 2])
    input_data = np.flip(input_data, axis=flip_axis)

    # Random Gaussian noise
    noise_std = random.uniform(0, 0.1)
    input_data += np.random.normal(0, noise_std, input_data.shape)

    return input_data


def augment_and_save_files(input_dir, output_dir):
    for category in os.listdir(input_dir):
        input_category_dir = os.path.join(input_dir, category)
        if os.path.isdir(input_category_dir):
            output_category_dir = os.path.join(output_dir, category)
            os.makedirs(output_category_dir, exist_ok=True)

            for filename in os.listdir(input_category_dir):
                if filename.endswith(".nii"):
                    file_path = os.path.join(input_category_dir, filename)

                    # Load NIfTI file
                    img = nib.load(file_path)
                    data = img.get_fdata()

                    # Extract filename without extension
                    file_name = os.path.splitext(filename)[0]

                    # Save original file
                    output_filename_original = os.path.join(output_category_dir, "original_" +
file_name + ".nii")
                    original_img = nib.Nifti1Image(data, img.affine, img.header)
                    nib.save(original_img, output_filename_original)

                    # Apply data augmentation
                    augmented_data = augment_data(data)

                    # Save augmented file
                    output_filename_augmented = os.path.join(output_category_dir, "augmented_" +
file_name + ".nii")
                    augmented_img = nib.Nifti1Image(augmented_data, img.affine, img.header)
                    nib.save(augmented_img, output_filename_augmented)


# Set the directories
train_data_dir = r"E:\PFG\DataDirectory\split_data_cropped_augmented_nifti_128x128x64\train"
output_dir = r"E:\PFG\DataDirectory\split_data_cropped_augmented_nifti_128x128x64_augmented"

# Create the output directory if it doesn't exist
os.makedirs(output_dir, exist_ok=True)

# Perform data augmentation and save original and augmented files
augment_and_save_files(train_data_dir, output_dir)

import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import nibabel as nib
from sklearn.model_selection import train_test_split
import os
import pickle

# Check if a GPU is available
if tf.config.list_physical_devices('GPU'):
```

```
    print("Running on GPU")
else:
    print("Running on CPU")

# Load NIfTI data and preprocess it
def load_nifti_data(file_path):
    img = nib.load(file_path)
    data = img.get_fdata()
    return data

# Set the paths to your NIfTI data
train_data_dir = r"E:\PFG\DataDirectory\split_data_cropped_augmented_nifti_128x128x64_augmented\train"
val_data_dir = r"E:\PFG\DataDirectory\split_data_cropped_augmented_nifti_128x128x64_augmented\val"
test_data_dir = r"E:\PFG\DataDirectory\split_data_cropped_augmented_nifti_128x128x64_augmented\test"

# Load and preprocess the training data
train_data = []
train_labels = []
train_dirs = ["healthy", "tumorous"]  # Subfolders within the train directory
for train_dir in train_dirs:
    dir_path = os.path.join(train_data_dir, train_dir)
    train_files = os.listdir(dir_path)
    for file in train_files:
        try:
            file_path = os.path.join(dir_path, file)
            if train_dir == "healthy":
                patient_number = file.split(".")[0][-3:]  # Extract the patient number from the file
name
            else:
                patient_number = file.split(".")[0][-5:]  # Extract the patient number from the file
name
            data = load_nifti_data(file_path)
            # Preprocess the data as needed

            # Reshape the data to match the input shape
            data = np.expand_dims(data, axis=-1)  # Add the grayscale channel dimension

            train_data.append(data)
            train_labels.append(0 if train_dir == "healthy" else 1)
        except Exception as e:
            print(f"Error: Failed to load {file}. Error message: {str(e)}")

# Load and preprocess the validation data
val_data = []
val_labels = []
val_dirs = ["healthy", "tumorous"]  # Subfolders within the validation directory
for val_dir in val_dirs:
    dir_path = os.path.join(val_data_dir, val_dir)
    val_files = os.listdir(dir_path)
    for file in val_files:
        try:
            file_path = os.path.join(dir_path, file)
            if val_dir == "healthy":
                patient_number = file.split(".")[0][-3:]  # Extract the patient number from the file
name
            else:
                patient_number = file.split(".")[0][-5:]  # Extract the patient number from the file
name
            data = load_nifti_data(file_path)
            # Preprocess the data as needed

            # Reshape the data to match the input shape
            data = np.expand_dims(data, axis=-1)  # Add the grayscale channel dimension

            val_data.append(data)
            val_labels.append(0 if val_dir == "healthy" else 1)
        except Exception as e:
            print(f"Error: Failed to load {file}. Error message: {str(e)}")


# Convert the data and labels to numpy arrays
train_data = np.array(train_data)
train_labels = np.array(train_labels)
val_data = np.array(val_data)
```

```
val_labels = np.array(val_labels)

seed = 99


# Define the model architecture
model = keras.Sequential(
    [
        layers.Input(shape=train_data[0].shape),  # Add the channel dimension
        layers.Conv3D(32, kernel_size=(3, 3, 3), activation="relu"),
        layers.MaxPooling3D(pool_size=(2, 2, 2)),
        layers.Conv3D(64, kernel_size=(3, 3, 3), activation="relu"),
        layers.MaxPooling3D(pool_size=(2, 2, 2)),
        layers.Conv3D(128, kernel_size=(3, 3, 3), activation="relu"),
        layers.MaxPooling3D(pool_size=(2, 2, 2)),
        layers.Flatten(),
        layers.Dense(64, activation="relu"),
        layers.Dropout(0.5),
        layers.Dense(1, activation="sigmoid"),
    ]
)

# Compile the model
model.compile(optimizer="adam", loss="binary_crossentropy", metrics=["accuracy"])


Running on GPU


# Train the model
with tf.device("CPU"):
    history = model.fit(train_data, train_labels, batch_size=8, epochs=10, validation_data=(val_data,
val_labels))

# Save the model
model.save('modeloaumentado.h5')

# Save the history as a file
with open('historymodeloaumentado.pickle', 'wb') as f:
    pickle.dump(history.history, f)

import numpy as np
import tensorflow as tf
from tensorflow import keras
import nibabel as nib
import os

# Check if a GPU is available
if tf.config.list_physical_devices('GPU'):
    print("Running on GPU")
else:
    print("Running on CPU")

# Load NIfTI data and preprocess it
def load_nifti_data(file_path):
    img = nib.load(file_path)
    data = img.get_fdata()
    return data

# Set the path to your test data directory
test_data_dir = r"E:\PFG\DataDirectory\split_data_cropped_augmented_nifti_128x128x64_augmented\test"

# Load and preprocess the test data
test_data = []
test_labels = []
test_dirs = ["healthy", "tumorous"]  # Subfolders within the test directory
for test_dir in test_dirs:
    dir_path = os.path.join(test_data_dir, test_dir)
    test_files = os.listdir(dir_path)
    for file in test_files:
        try:
            file_path = os.path.join(dir_path, file)
            if test_dir == "healthy":
                patient_number = file.split(".")[0][-3:]  # Extract the patient number from the file
```

```
name
            else:
                patient_number = file.split(".")[0][-5:]  # Extract the patient number from the file
name
            data = load_nifti_data(file_path)
            # Preprocess the data as needed

            # Reshape the data to match the input shape
            data = np.expand_dims(data, axis=-1)  # Add the grayscale channel dimension

            test_data.append(data)
            test_labels.append(0 if test_dir == "healthy" else 1)
        except Exception as e:
            print(f"Error: Failed to load {file}. Error message: {str(e)}")

# Convert the test data and labels to numpy arrays
test_data = np.array(test_data)
test_labels = np.array(test_labels)

# Load the saved model
model = keras.models.load_model("modeloaumentado.h5")

# Set the device to CPU for evaluation
with tf.device("CPU"):
    # Evaluate the model on the test set
    loss, accuracy = model.evaluate(test_data, test_labels)
    print(f"Test loss: {loss}")
    print(f"Test accuracy: {accuracy}")
```

This code snippet loads a history file containing training and validation metrics and plots the training accuracy, training loss, validation accuracy, and validation loss using matplotlib.

```
import pickle
import matplotlib.pyplot as plt

# Specify the path to the directory where the history file is located
history_dir = r"E:\PFG\Resultados"

# Load the history from file
history_file = os.path.join(history_dir, "historymodeloaumentado.pickle")
with open(history_file, 'rb') as f:
    history = pickle.load(f)

# Plot the training and validation metrics
fig, ax = plt.subplots(1, 2, figsize=(20, 3))
ax = ax.ravel()

for i, metric in enumerate(["accuracy", "loss"]):
    ax[i].plot(history[metric])
    ax[i].plot(history["val_" + metric])
    ax[i].set_title("Model {}".format(metric))
    ax[i].set_xlabel("Epochs")
    ax[i].set_ylabel(metric)
    ax[i].legend(["Train", "Validation"])

plt.show()
```

This code snippet demonstrates how to make predictions on a single scan using two different models and plot slices of the scan data.

```
import random
import matplotlib.pyplot as plt

# Assuming you have loaded and compiled the first model previously
model.load_weights("2modelo.h5")  # Load the saved weights

# Assuming you have loaded and compiled the second model previously
second_model = keras.models.load_model("modeloaumentado.h5")

# Select a random scan index from the test dataset
scan_index = random.randint(0, len(test_data)-1)
```

```
# Make predictions on a single scan using the first model
scan_data = test_data[scan_index]  # Get the data of the scan

# Reshape the scan data if necessary (assuming the shape is (128, 128, 64, 1))
scan_data = scan_data.squeeze(axis=-1)

prediction = model.predict(np.expand_dims(scan_data, axis=0))[0]
scores = [1 - prediction[0], prediction[0]]

class_names = ["healthy", "tumorous"]
for score, name in zip(scores, class_names):
    print(
        "First Model: This model is %.2f percent confident that the scan is %s"
        % ((100 * score), name)
    )

# Make predictions on the same scan using the second model
second_prediction = second_model.predict(np.expand_dims(scan_data, axis=0))[0]
second_scores = [1 - second_prediction[0], second_prediction[0]]

for score, name in zip(second_scores, class_names):
    print(
        "Second Model: This model is %.2f percent confident that the scan is %s"
        % ((100 * score), name)
    )

# Plot slices of the scan data
num_slices = scan_data.shape[-1]
slices_to_plot = [int(num_slices/4), int(num_slices/2), int(3*num_slices/4)]  # Choose slices to plot

fig, ax = plt.subplots(1, len(slices_to_plot), figsize=(15, 5))
for i, slice_index in enumerate(slices_to_plot):
    ax[i].imshow(scan_data[:, :, slice_index], cmap="gray")
    ax[i].axis("off")
    ax[i].set_title("Slice {}".format(slice_index+1))

plt.tight_layout()
plt.show()
```

This code snippet performs the evaluation of a trained model on the test set and plots the Receiver Operating Characteristic (ROC) curve.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
import tensorflow as tf
from tensorflow import keras
import nibabel as nib
import os

# Check if a GPU is available
if tf.config.list_physical_devices('GPU'):
    print("Running on GPU")
else:
    print("Running on CPU")

# Load NIfTI data and preprocess it
def load_nifti_data(file_path):
    img = nib.load(file_path)
    data = img.get_fdata()
    return data

# Set the path to your test data directory
test_data_dir = r"E:\PFG\DataDirectory\split_data_cropped_augmented_nifti_128x128x64_augmented\test"

# Load and preprocess the test data
test_data = []
test_labels = []
test_dirs = ["healthy", "tumorous"]  # Subfolders within the test directory
for test_dir in test_dirs:
    dir_path = os.path.join(test_data_dir, test_dir)
```

```
    test_files = os.listdir(dir_path)
    for file in test_files:
        try:
            file_path = os.path.join(dir_path, file)
            if test_dir == "healthy":
                patient_number = file.split(".")[0][-3:]  # Extract the patient number from the file
name
            else:
                patient_number = file.split(".")[0][-5:]  # Extract the patient number from the file
name
            data = load_nifti_data(file_path)
            # Preprocess the data as needed

            # Reshape the data to match the input shape
            data = np.expand_dims(data, axis=-1)  # Add the grayscale channel dimension

            test_data.append(data)
            test_labels.append(0 if test_dir == "healthy" else 1)
        except Exception as e:
            print(f"Error: Failed to load {file}. Error message: {str(e)}")

# Convert the test data and labels to numpy arrays
test_data = np.array(test_data)
test_labels = np.array(test_labels)

# Load the saved model
model = keras.models.load_model("modeloaumentado.h5")

# Set the device to CPU for evaluation
with tf.device("CPU"):
    # Evaluate the model on the test set
    loss, accuracy = model.evaluate(test_data, test_labels)
    print(f"Test loss: {loss}")
    print(f"Test accuracy: {accuracy}")

    # Make predictions on the test set
    predictions = model.predict(test_data)

# Compute the false positive rate (fpr), true positive rate (tpr), and thresholds
fpr, tpr, thresholds = roc_curve(test_labels, predictions)

# Compute the area under the ROC curve (AUC)
roc_auc = auc(fpr, tpr)

# Plot the ROC curve
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()
```

Annex C: Budget

EQUIPMENT BUDGET

Usage time (hours)	Reference	Description	Price (€)	
			Average Amortization Cost (per hour)	Total
350	HP Victus 16-e0090ns	Computer	3.10	1088.00
Total Equipment				1088.00

SOFTWARE BUDGET

Usage time (hours)	Reference	Description	Price (€)	
			Average Amortization Cost (per hour)	Total
350	-	Licence Python	0	0.0
Total software				0.0

TRAINING BUDGET

Usage time (hours)	Reference	Description	Price (€)	
			Unitary	Total
35	PCEP-30-xx	Certified Entry-Level Python Programmer	84.90	84.90
40	PCAP-31-xx	Certified Associate in Python Programming	319.00	319.00
45	PCAD-31-xx	Certified Associate in Data Analysis with Python	319.00	319.00
Total				722.90

LABOR COST BUDGET

Task	Duration (hours)	Price (€)	
		Unitary	Total
Student Dedication	375	25.00	9375
Total Labor Cost			9375

BUDGET SUMMARY

Category	Amount (€)	
	Partial	Acumulated
Training	722.90	722.90
Consumables	0.00	722.90
Equipment	1088.00	1810.90
Software	0.0	1810.90
Labor Cost	9375.00	11185.90
Indirect Costs (10%)		1118.59
Total without VAT		12304.49
Total with VAT		14888.43