Design, FPGA Implementation and Analysis of a CNN-Based Tumor Detection System

Project report submitted in partial fulfillment
of the requirements for the degree of

Bachelor of Technology
in
Electronics and Communication Engineering

by

Tanmay Rawal
Roll No-23DEC511

Under Guidance of

Dr. Kusum Lata
Professor
Electronics and Communication Engineering

Department of Electronics and Communication Engineering
The LNM Institute of Information Technology, Jaipur

December 7, 2025

Acknowledgment

We present my heartfelt gratitude to my project Supervisor, **Dr. Kusum Lata** (Professor, Electronics and Communication Engineering), for giving me the opportunity to work under her guidance on this project titled **"Design, FPGA Implementation and Analysis of a CNN-Based Tumor Detection System"** and for encouraging me to give my best to the project.

I would also like to thank the respected PhD scholar **Prashant Singh** from the bottom of my heart for his constant guidance and support throughout the project, for patiently solving my doubts, and for introducing me to the Hardware.

Abstract

Early and accurate tumor detection is essential for improving diagnostic outcomes and supporting clinical decision-making. This project presents a complete software–hardware pipeline for designing, analyzing, and deploying a CNN-based tumor detection system on FPGA hardware. Three deep-learning architectures—EfficientNetB0, Inception, and Xception—were trained and evaluated both with and without data augmentation to compare their performance in terms of accuracy and key classification metrics. All training, preprocessing, and evaluation were conducted in a GPU-enabled Jupyter environment to ensure efficient experimentation and faster convergence. Furthermore, model interpretability was explored using Grad-CAM, LIME, and SHAP to visualize tumor regions and understand the feature importance learned by each network.

To prepare the trained models for hardware deployment, quantization was performed using a Docker-based workflow with the Vitis AI image, enabling the conversion of floating-point models into hardware-friendly fixed-point formats. The optimized model was then compiled and deployed on the PYNQ-ZU FPGA platform, providing real-time inference under low-power constraints. Experimental results highlight the trade-offs among the different architectures and demonstrate that the FPGA-accelerated implementation achieves a balanced combination of accuracy, speed, and resource efficiency. This work illustrates the effectiveness of integrating deep-learning interpretability, model optimization, and FPGA acceleration to develop deployable, high-performance tumor detection systems suitable for edge-level medical imaging applications.

Contents

1 Introduction

Brain tumors represent one of the most severe and life-threatening neurological conditions, where early and accurate diagnosis plays a critical role in improving patient outcomes. Medical imaging, particularly Magnetic Resonance Imaging (MRI), is the primary non-invasive modality used for detecting and characterizing brain tumors. In recent years, deep learning—especially Convolutional Neural Networks (CNNs)—has emerged as a powerful tool for automating and improving brain tumor classification and detection. However, deploying deep-learning models in real clinical or edge environments remains challenging due to high computational demands, limited interpretability, and the need for real-time performance in resource-constrained systems. This project addresses these challenges by developing a complete end-to-end brain tumor detection system, covering software training, explainability, optimization, and FPGA-based hardware deployment.

To thoroughly evaluate deep-learning performance for brain tumor detection, three CNN architectures—**EfficientNetB0**, **Inception**, and **Xception**—were selected based on their efficiency and proven accuracy in medical image classification tasks. Each model was trained under two configurations: with data augmentation, allowing a detailed comparison of generalization ability and robustness to variations commonly observed in MRI scans. Training and experimentation were conducted in a GPU-enabled Jupyter environment, enabling efficient preprocessing, rapid model convergence, and large-scale metric evaluation. Standard performance indicators such as accuracy, precision, recall, F1-score, and confusion matrices were used to benchmark the models.

Given the critical importance of interpretability in medical AI systems, this project integrates three complementary explainability techniques—**Grad-CAM**, **LIME**, and **SHAP**—to visualize brain tumor regions and understand the internal decision-making process of the CNN models. Grad-CAM highlights key areas within MRI scans that influence model predictions, while LIME and SHAP provide more granular, pixel-level feature attribution, ensuring transparent and clinically meaningful explanations.

To transition from software development to deployable hardware, the trained models underwent quantization, converting high-precision floating-point weights into low-precision fixed-point representations. This optimization significantly reduces model size, latency, and computational cost—factors essential for real-time embedded applications. Quantization and compilation were performed using a Docker-based environment running the Vitis AI toolchain, ensuring compatibility with Xilinx's deployment workflow. The optimized model was then implemented on the **PYNQ-ZU FPGA platform**, leveraging its programmable logic fabric and ARM processing system to deliver high-speed, low-power inference suitable for edge-level medical imaging devices.

Overall, this project delivers a complete and optimized brain tumor detection system that integrates deep-learning model development, interpretability, hardware-aware optimization, and real-time FPGA execution. By combining multiple CNN architectures, advanced explainability methods, and efficient hardware acceleration, the work demonstrates a practical pathway for transforming deep-learning-based brain tumor classifiers into deployable and clinically relevant solutions.

1.1 The Brain Tumor MRI Dataset

The Kaggle-sourced brain tumor MRI dataset used in this project consists of a comprehensive collection of 7023 MRI scans. These images are classified into four categories: **Glioma tumors**, **Meningioma tumors**, **Pituitary tumors**, and **No-tumor** images . This class distribution provides a diverse representation of brain tumor types. Figure illustrates an example of the dataset composition.

The dataset incorporates images from multiple well-known sources, including Figshare, the SARTAJ dataset, and the Br35H dataset, ensuring a rich and heterogeneous variety of MRI scans suitable for robust model training and evaluation.

The dataset is divided into training and testing subsets. The **training set** includes:

- 1457 Pituitary tumor images
- 1339 Meningioma tumor images
- 1321 Glioma tumor images
- 1595 No-tumor images

The **testing set** contains:

- 300 Pituitary tumor images
- 306 Meningioma tumor images
- 300 Glioma tumor images
- 405 No-tumor images

This balanced division enables consistent performance evaluation across all tumor categories and supports generalization in real-world medical imaging applications.

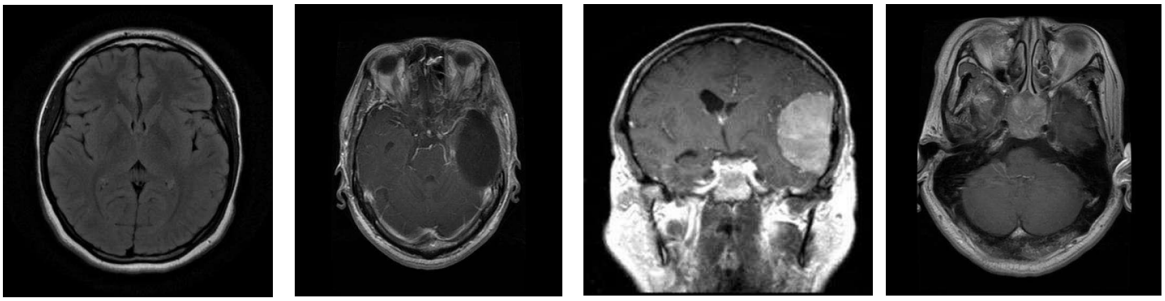


Figure 1: Overview of the brain tumor MRI dataset showing the four classes: No Tumor, Glioma, Meningioma, and Pituitary.

1.2 Preprocessing Techniques for the Brain Tumor MRI Dataset

Effective image preprocessing is crucial for enhancing the performance of deep learning models, particularly in medical imaging tasks. This study systematically applies several preprocessing techniques to the Brain Tumor MRI Dataset to prepare the images for training and improve model accuracy. The preprocessing steps include image cropping, noise removal, colormap application, and resizing. Each step plays a significant role in refining the images, ensuring that the learning algorithms focus on relevant features and that the dataset is standardized for consistent model input.

1.2.1 Cropping the Images

The first step in preprocessing involves cropping the MRI images to eliminate unwanted background noise commonly present in raw scans. Background noise may include non-brain structures or artifacts that are irrelevant for tumor detection. By cropping the images, the algorithm focuses solely on the brain region where tumors may exist, thereby improving feature extraction accuracy. This isolation of the region of interest (ROI) ensures that the model is not distracted by irrelevant regions, resulting in higher-quality input data for learning.

Figure 4 illustrates the cropping process through four distinct steps:

- **Step 1: Obtain the Original Image** — The initial MRI scan captures the entire head, including the brain and surrounding tissues.
- **Step 2: Find the Biggest Contour** — The largest contour (shown in blue) is detected around the brain area, identifying the primary object of interest.
- **Step 3: Identify the Extreme Points** — Extreme points (red, blue, green, and yellow dots) on the boundaries of the brain are extracted to assist in determining the cropping boundaries.
- **Step 4: Crop the Image** — The final cropped image isolates the brain region while removing extraneous areas such as hair and skin, allowing more focused analysis for tumor detection.

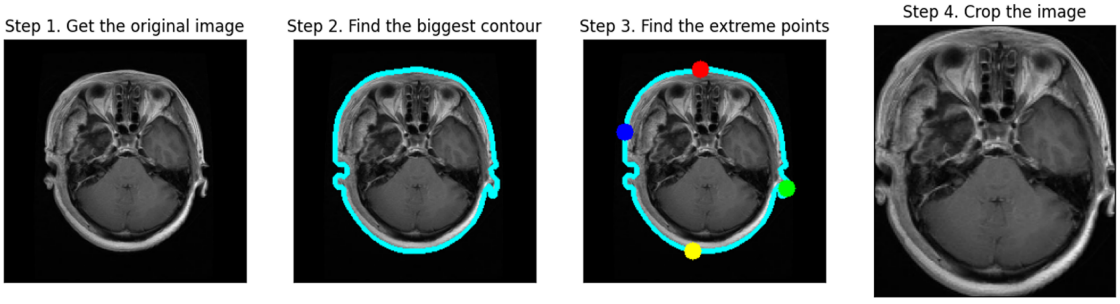


Figure 2: Illustration of the MRI image cropping process.

1.2.2 Noise Removal

After cropping, noise removal is performed using a bilateral filter. The bilateral filter is particularly effective for MRI scans because it smooths the image while preserving edges and fine structural details. Unlike traditional filters that blur important anatomical boundaries along with noise, the bilateral filter considers both spatial distance and intensity similarity between pixels. This dual consideration allows the filter to reduce noise while maintaining edge integrity, which is essential for identifying tumor boundaries.

1.2.3 Applying Colormap

The next preprocessing step involves applying a colormap to the images. Colormap application enhances the interpretability of MRI scans by improving contrast between different tissues or anatomical structures in the brain. In raw grayscale MRI scans, subtle differences in tissue intensity may be difficult to distinguish. A colormap enhances these variations by mapping pixel intensity values to colors, thereby improving visual separation between tumor and non-tumor regions.

1.2.4 Resize

The final preprocessing step involves resizing the images, which is crucial for standardizing input dimensions for deep learning models. Convolutional neural networks require input images of a fixed size. Resizing ensures that all images in the dataset conform to the required dimensions, facilitating consistent processing and analysis.

Standardizing image size also reduces computational complexity and memory usage, enabling more efficient model training. During resizing, special care is taken to preserve the aspect ratio of the images to avoid distortion, which could compromise important features. .

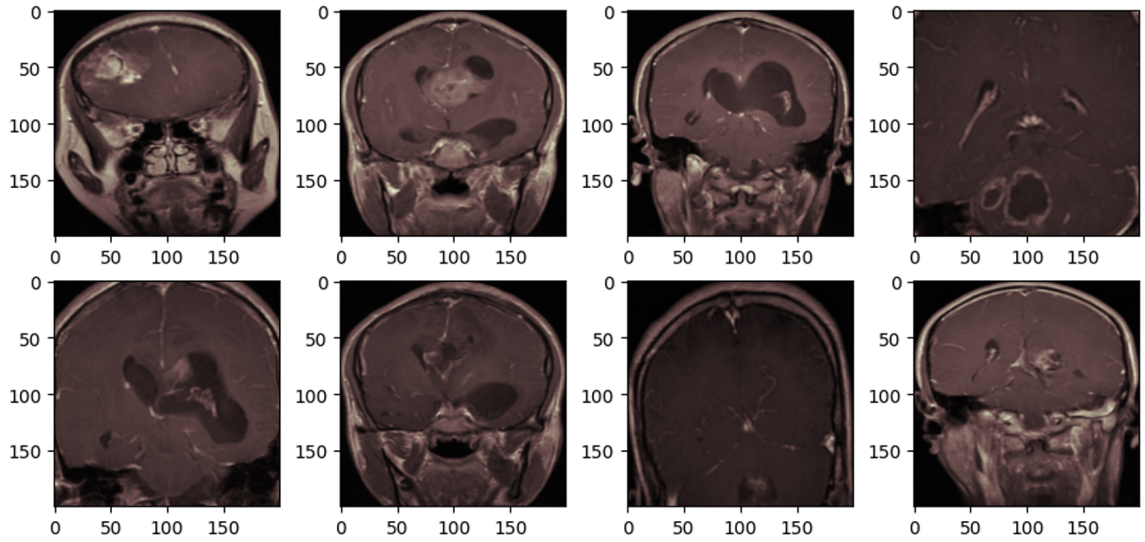


Figure 3: Brain Tumor MRI after Preprocessing.

1.3 Data Partitioning: Training, Validation, and Testing Sets

For this experiment, the dataset was divided into training, validation, and testing subsets using an effective and balanced data partitioning strategy. A standard 80–10–10 split was adopted to ensure that the model was trained on a sufficiently large number of samples while maintaining reliable evaluation and validation performance.

The entire set of training images provided in the original dataset was used exclusively for training, resulting in a total of **5712 training images**. For the remaining images located in the testing folder, a custom split was applied:

- **50% of the testing folder** was used to create the **validation set**, resulting in **656 images**.
- The remaining **50%** of the testing folder was retained as the final **testing set**, containing **655 images**.

1.4 Image Augmentation

The limited size of medical image datasets poses a significant challenge in achieving robust performance in deep learning models. To overcome this limitation, effective utilization of image augmentation techniques becomes essential. Image augmentation artificially increases the diversity of the training dataset by applying various transformations to existing images, thereby improving the model's ability to generalize.

In this study, `ImageDataGenerator` provided by the Keras deep learning library was used to perform real-time data augmentation during training. This tool applies a range of transformations such as rotations, translations, and flips to generate new image variations on the fly. Such augmentation not only expands the effective size of the training set but also helps mitigate overfitting, particularly when dealing with relatively small medical imaging datasets.

The experimental configuration used the following augmentation parameters:

- **rotation_range = 10** — Applies random rotations up to 10 degrees.
- **width_shift_range = 0.05** — Introduces horizontal translations up to 5% of the image width.
- **height_shift_range = 0.05** — Introduces vertical translations up to 5% of the image height.
- **horizontal_flip = True** — Enables random horizontal flipping.

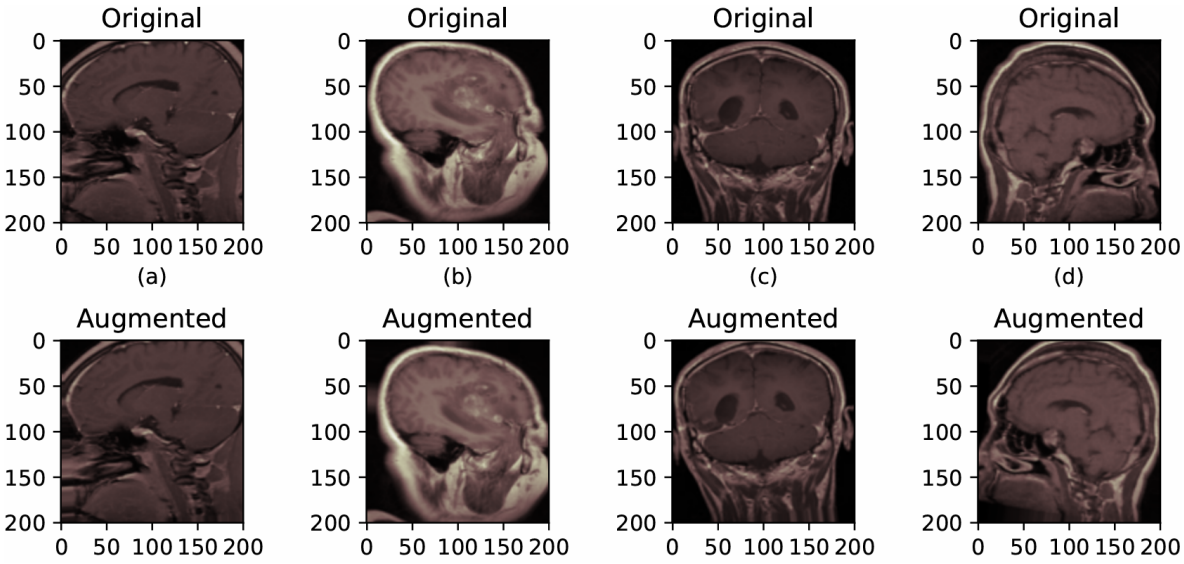


Figure 4: MRI after Augmentation.

.

2 Deep Learning Models: Design and Implementation

2.1 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) represent a foundational deep learning architecture widely used in image classification tasks. These networks excel at automatically learning hierarchical patterns from raw image data with minimal preprocessing. Over time, CNNs have significantly evolved, incorporating advanced techniques such as transfer learning, fine-tuning, and layer freezing, which have substantially improved their performance beyond that of traditional machine learning models .

A CNN is typically composed of three essential types of layers, each performing a distinct function within the overall architecture:

- **Convolutional Layer:** Applies multiple kernels (filters) that slide across the input image or feature map, extracting spatial features such as edges, textures, and shapes. These feature maps capture different aspects of the image's spatial hierarchy.
- **Pooling Layer:** Often placed after convolutional layers, pooling layers downsample feature maps to reduce spatial dimensions and network parameters.
- **Fully Connected Layer:** Flattens the extracted features into a vector and processes them through dense layers to perform final classification. These layers typically contain a large number of parameters, making them computationally intensive during training.

In this study, three deep learning models—**EfficientNetB0**, **InceptionV3**, **Xception**, **Ensemble Model**,—were selected for brain tumor classification using MRI scans. These models were chosen due to their proven effectiveness and widespread adoption in medical image classification tasks.

2.1.1 InceptionV3

InceptionV3 is a widely adopted Convolutional Neural Network architecture designed specifically for large-scale image classification tasks. It enhances the original Inception module by incorporating factorized convolutions, dimensionality reduction techniques, and optimized computational efficiency. The architecture includes multiple blocks composed of convolutional layers, pooling layers, and fully connected layers, enabling hierarchical feature extraction at various spatial scales.

To address overfitting, InceptionV3 integrates dropout layers and batch normalization throughout its structure. The network consists of 42 layers and contains approximately 21.8 million parameters, making it a powerful and efficient choice for extracting complex patterns in medical imaging. Due to its depth, architectural optimizations, and strong generalization capabilities, InceptionV3 is frequently used in tumor detection and other biomedical image analysis applications.

2.1.2 Xception

Xception, a convolutional neural network (CNN) architecture introduced by Chollet in 2017, integrates pointwise convolution and depthwise separable convolution. It comprises 71 layers organized into three main flows: the entry flow, the middle flow, and the exit flow. Unlike conventional architectures, Xception adopts a unique approach where convolution is not conducted across all channels simultaneously. This strategic modifica tion reduces interconnections and effectively reduces the total number of parameters to approximately 21 million.

2.1.3 EfficientNetB0

EfficientNetB0 is the baseline model of the EfficientNet family, developed using a compound scaling method that uniformly scales depth, width, and resolution to achieve an optimal balance between accuracy and computational efficiency. Unlike traditional architectures that scale network dimensions arbitrarily, EfficientNet employs a principled scaling strategy that enhances performance while maintaining low computational cost. With approximately 5.3 million parameters, EfficientNetB0 is significantly lighter than conventional deep CNNs, making it highly suitable for real-time and resource-constrained environments.

2.1.4 Ensemble Model

To enhance robustness and classification performance, an Ensemble Model was constructed by combining the outputs of the three best-performing architectures: EfficientNetB0, InceptionV3, and Xception. Ensemble learning leverages the complementary strengths of multiple models, thereby reducing individual model biases and improving generalization across diverse MRI samples.

In this study, the ensemble prediction was generated using a **weighted averaging** approach. Each model produced a probability distribution over the four classes, and the final prediction was obtained by computing a weighted average of these probabilities. The weights were empirically chosen based on validation performance, ensuring that stronger models contributed more significantly to the final output.

2.2 Leveraging Transfer Learning with ImageNet Pre-Training

Transfer learning is a powerful deep learning technique that enables models to utilize knowledge gained from large-scale datasets and apply it to specialized tasks with limited data. Instead of training a network from scratch, the model begins with pre-trained weights that already encode rich feature representations. This approach significantly accelerates convergence, improves generalization, and reduces the amount of data required for effective training.

In this study, transfer learning was employed by initializing EfficientNetB0, InceptionV3, and Xception with **ImageNet pre-trained weights**. ImageNet, a large dataset containing over 1.2 million natural images across 1000 categories, provides robust low-level and mid-level feature extraction capabilities such as edge detection, texture recognition, and shape identification. These foundational features transfer well to medical imaging tasks, including MRI-based brain tumor classification.

By fine-tuning the pre-trained layers and training the deeper layers on the MRI dataset, the models adapt these generalized visual features to the specific patterns associated with brain tumors. This strategy leads to improved accuracy, reduced overfitting, and more efficient learning compared to models trained entirely from scratch.

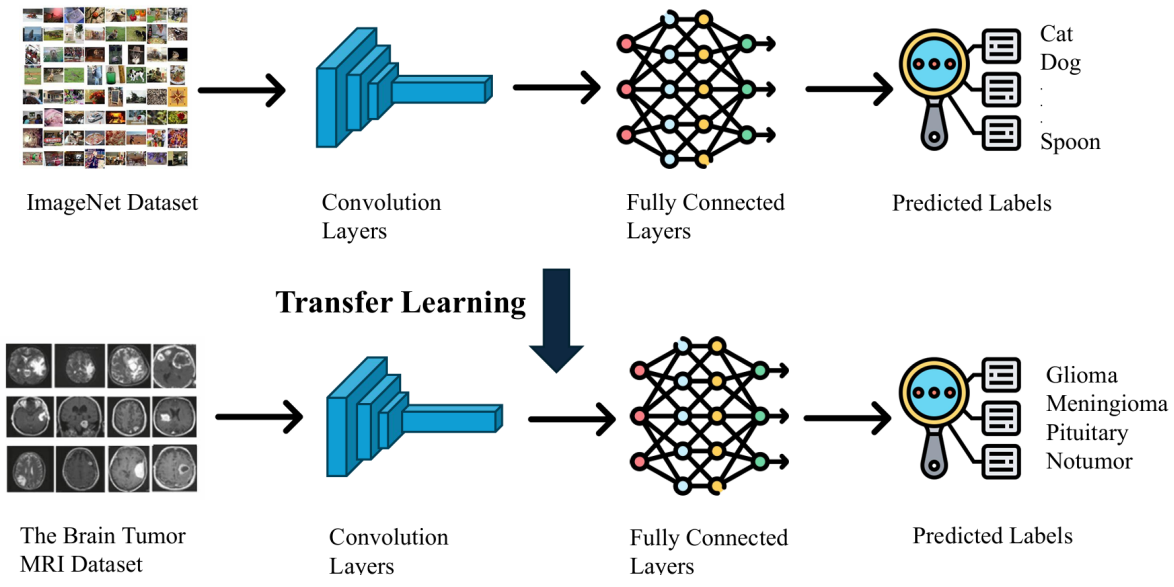


Figure 5: Explaining Transfer Learning.

2.3 Optimizing Brain Tumor Classification: Training and Tuning Strategies

Hyperparameter tuning further enhances the model's performance after pre-training on the ImageNet dataset. Hyperparameters are predefined settings that govern the learning process and are not learned from the data itself. These include parameters such as the learning rate, batch size, number of epochs, optimizer type, and dropout rates. Selecting optimal hyperparameters is critical, as they directly influence the model's convergence behavior, generalization ability, and overall classification accuracy.

Table 1: Hyperparameter Configuration Used for Model Training

Hyperparameter	**Configuration**
Learning Rate	1×10^{-4}
Mini-Batch Size	32 Images
Maximum Epochs	25
Dropout Probability	0.4
Optimization Algorithm	Adam
Activation Function (Final Layer)	Softmax
Loss Function	Categorical Cross-Entropy

2.4 Performance Evaluation Metrics

To comprehensively assess the performance of the deep learning models used for brain tumor classification, several evaluation metrics were employed. Confusion matrices (CM) provided detailed insights into model predictions by presenting the counts of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). Based on the confusion matrix, key evaluation metrics such as accuracy, precision, recall, and F1-score were computed to capture different dimensions of classification performance.

2.4.1 Accuracy

Accuracy measures the overall correctness of the model's predictions. It is defined as the ratio of correctly classified instances to the total number of predictions. In multi-class brain tumor detection, high accuracy indicates that the model is effective across various tumor types. However, accuracy alone may not be sufficient for imbalanced datasets.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

2.4.2 Precision

Precision evaluates the proportion of positive predictions that are actually correct. High precision reduces the risk of false positives, which is critically important in medical diagnosis to avoid misclassification of healthy tissue as a tumor.

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

2.4.3 Recall (Sensitivity)

Recall, also known as sensitivity, measures the model's ability to correctly identify true positive cases. High recall ensures that most tumor cases—especially malignant ones—are detected early, reducing the chances of missed diagnoses.

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

2.4.4 F1-Score

The F1-score is the harmonic mean of precision and recall, providing a balanced measure that accounts for both false positives and false negatives. This metric is especially useful when dealing with uneven class distributions.

$$F1 - Score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{4}$$

In addition to classification metrics, the model's inference time was recorded to assess real-time applicability. Training time was also measured to evaluate the efficiency of retraining models when new data is introduced. These measurements support both offline and online learning paradigms in adaptive healthcare systems.

3 Software Results

This section presents the software-side results obtained from training and evaluating the deep learning models for brain tumor classification. The performance of EfficientNetB0, InceptionV3, Xception, and the Ensemble model is analyzed in terms of accuracy, precision, recall, F1-score, and confusion matrices. Additional visualizations such as training/validation loss and accuracy curves, along with explainability plots (Grad-CAM, LIME, and SHAP), are also discussed.

3.1 Comparison of Different Models in Terms of Classification Metrics

This subsection provides a detailed comparison of the four deep learning models employed in the software implementation—EfficientNetB0, InceptionV3, Xception, and the proposed Weighted Ensemble—using key evaluation metrics, namely class-wise precision, recall, F1-score, and overall accuracy, as summarized in Table 2.

Table 2: Comparison of Different Models in Terms of Classification Metrics

Model	Class	Precision	Recall	F1-score	Accuracy
EfficientNetB0	Glioma	1.0000	0.9467	0.9726	
	Meningioma	0.9497	0.9869	0.9679	
	Notumor	1.0000	1.0000	1.0000	
	Pituitary	0.9868	1.0000	0.9934	
	Weighted Avg	**0.9853**	**0.9848**	**0.9847**	**0.9848**
InceptionV3	Glioma	0.9932	0.9667	0.9797	
	Meningioma	0.9742	0.9869	0.9805	
	Notumor	1.0000	1.0000	1.0000	
	Pituitary	0.9868	1.0000	0.9934	
	Weighted Avg	**0.9894**	**0.9893**	**0.9893**	**0.9893**
Xception	Glioma	1.0000	0.9733	0.9865	
	Meningioma	0.9805	0.9869	0.9837	
	Notumor	1.0000	1.0000	1.0000	
	Pituitary	0.9804	1.0000	0.9901	
	Weighted Avg	**0.9910**	**0.9909**	**0.9908**	**0.9909**
Weighted Ensemble	Glioma	1.0000	0.9800	0.9899	
	Meningioma	0.9870	0.9935	0.9902	
	Notumor	1.0000	1.0000	1.0000	
	Pituitary	0.9868	1.0000	0.9934	
	Weighted Avg	**0.9940**	**0.9939**	**0.9939**	**0.9939**

Table 3: Comparison of Deep Learning Models Based on Classification Metrics

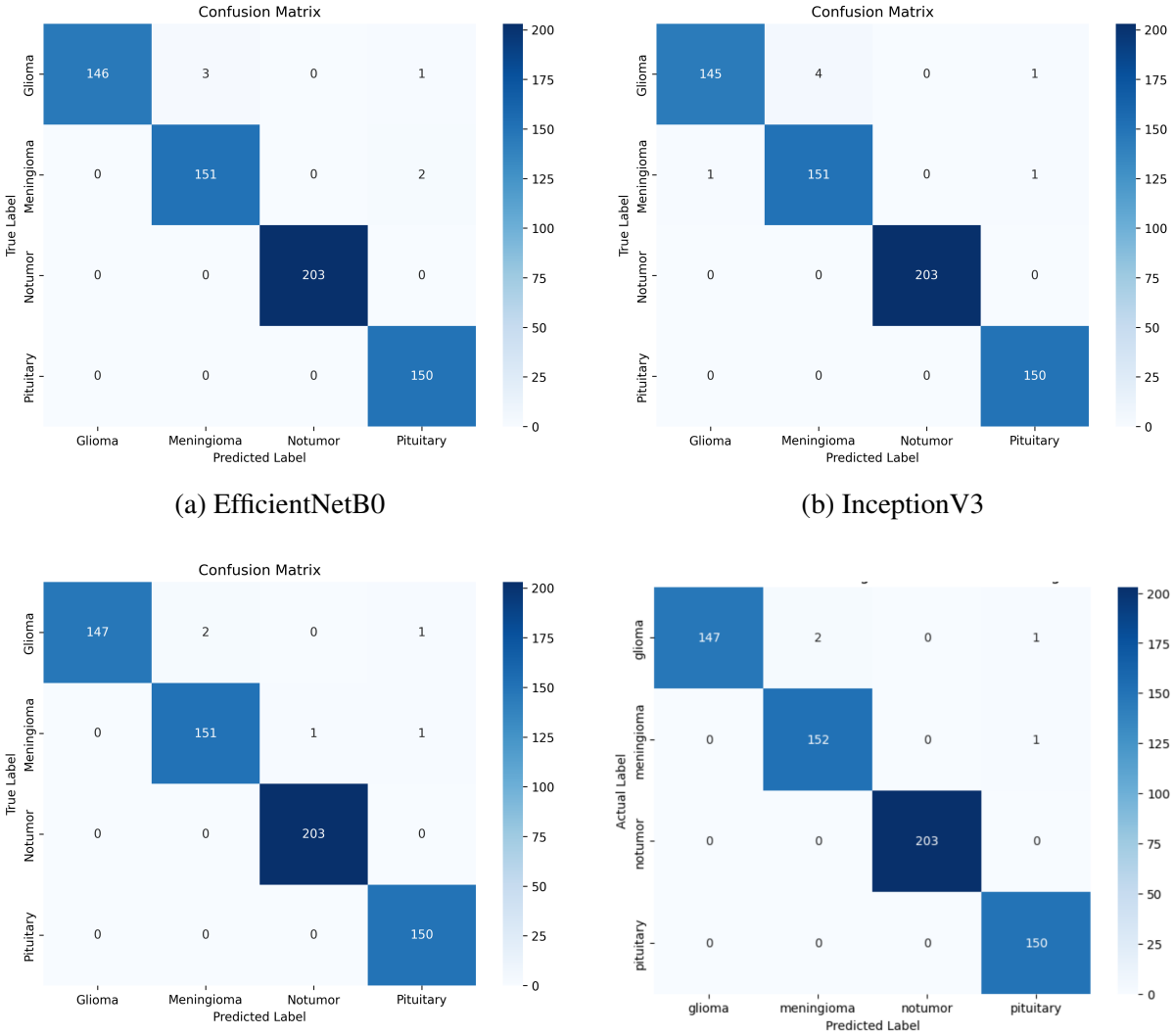
Model	Accuracy	Macro Avg F1	Weighted Avg F1	Remarks
EfficientNetB0	0.9848	0.9835	0.9847	Strong baseline performance; reliable across all tumor classes and suitable for latency-sensitive deployments.
InceptionV3	0.9893	0.9884	0.9893	Improved feature extraction and class-level consistency compared to EfficientNetB0, with only a modest increase in complexity.
Xception	0.9909	0.9901	0.9908	Best-performing individual model in terms of accuracy and F1-score; provides very strong generalization at higher computational cost.
Ensemble (Weighted)	0.9939	0.9933	0.9939	Highest overall performance and most stable predictions, achieved by combining all three base models; ideal when diagnostic precision is prioritized.

3.2 Confusion Matrix Analysis

The confusion matrix provides a detailed breakdown of the classification performance of the models by showing the number of correct and incorrect predictions for each tumor class. Unlike aggregated metrics such as accuracy or F1-score, the confusion matrix offers deeper insight into class-wise behavior, misclassification patterns, and the overall reliability of the model.

In this project, confusion matrices were generated for all four deep learning models—EfficientNetB0, InceptionV3, Xception, and the Weighted Ensemble. Each matrix illustrates how well the model distinguishes among the four brain tumor categories: *Glioma*, *Meningioma*, *Pituitary*, and *No tumor*.

A well-performing model exhibits strong diagonal dominance in the matrix, indicating that most samples are correctly classified. Misclassifications typically appear as off-diagonal values, which may highlight similarities between tumor structures or limitations in feature extraction.



(a) EfficientNetB0

(b) InceptionV3

(c) Xception

(d) Weighted Ensemble

Figure 6: Confusion Matrices for All Deep Learning Models

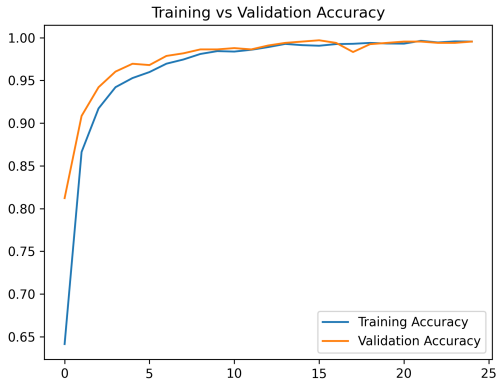
3.3 Training vs. Validation Accuracy

Training and validation accuracy curves were examined to evaluate how effectively each model learns discriminative features over the course of training and how well this learning generalizes to unseen data. As illustrated in Figure 7, all three architectures—EfficientNetB0, InceptionV3, and Xception—show a rapid increase in both training and validation accuracy during the initial epochs, followed by a gradual saturation close to 100% accuracy.

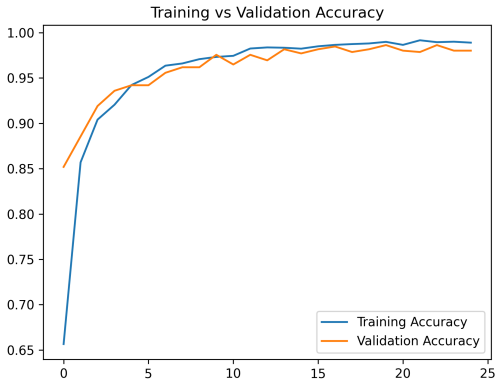
Across the models, the training and validation accuracy curves remain closely aligned, with only minor fluctuations in the validation accuracy at later epochs. This close tracking between the two curves indicates that the models do not suffer from severe overfitting; instead, they maintain strong generalization performance on the validation set. InceptionV3 and Xception reach slightly higher and more stable validation accuracy compared to EfficientNetB0, which is consistent with their superior classification metrics reported in the evaluation results.

Since the Weighted Ensemble model is obtained by combining the predictions of these three base models rather than training a separate network, it does not possess its own training or

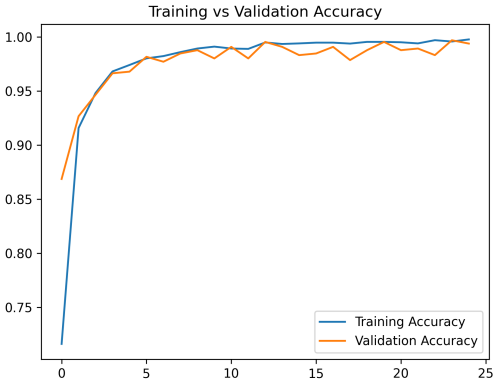
validation accuracy curve. Consequently, the accuracy curve analysis focuses exclusively on EfficientNetB0, InceptionV3, and Xception.



(a) EfficientNetB0 Accuracy



(b) InceptionV3 Accuracy



(c) Xception Accuracy

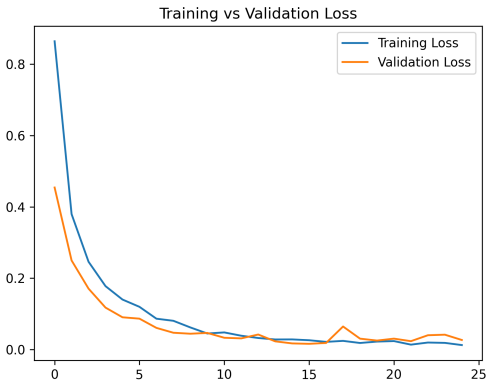
Figure 7: Training and Validation Accuracy Curves for Deep Learning Models

3.4 Training vs. Validation Loss

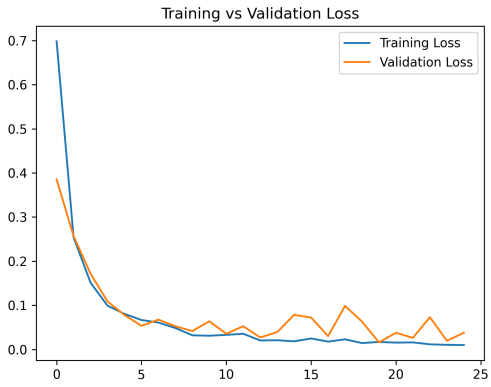
Training and validation loss curves were analyzed to study the optimization behavior and generalization capability of the models. As shown in Figure 8, all three architectures—EfficientNetB0, InceptionV3, and Xception—exhibit a rapid decrease in both training and validation loss during the initial epochs, followed by a gradual convergence towards low and stable values. This trend indicates that the models successfully learn discriminative features from the MRI data without suffering from severe optimization issues.

Across all three models, the training and validation loss curves remain relatively close to each other, with only small gaps observed toward later epochs. These minor differences, combined with the absence of a clear upward trend in validation loss, suggest that none of the models experiences significant overfitting. Instead, the loss curves reflect good generalization to unseen validation data. Among the models, Xception and InceptionV3 achieve slightly lower final validation loss, which is consistent with their superior classification performance reported in the evaluation metrics.

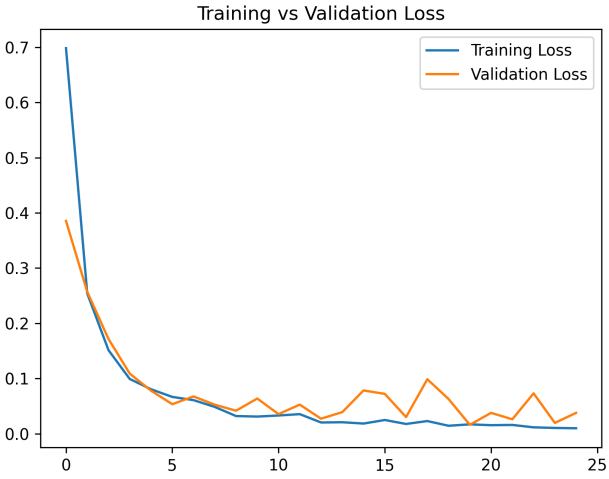
Since the Weighted Ensemble model is formed by aggregating the predictions of the individual base models rather than being trained end-to-end, it does not have its own training or validation loss curve. Therefore, the loss analysis is focused exclusively on the three constituent models.



(a) EfficientNetB0 Loss



(b) InceptionV3 Loss



(c) Xception Loss

Figure 8: Training and Validation Loss curves for Deep Learning Models

3.5 Prediction Time Across Different Models

The prediction time is a critical factor in the practical deployment of brain tumor classification models. In a clinical environment, faster predictions can support quicker decision-making and contribute to improved patient outcomes. Models with shorter inference times are therefore preferable for integration into medical imaging systems, where near real-time analysis is often required.

Table 4 summarizes the minimum, maximum, and average prediction times (in milliseconds) for each model on the entire test set. Among these measurements, the average prediction time is the most informative, as it reflects the typical latency observed during routine use.

The results show that **EfficientNetB0** achieves the lowest average prediction time of approximately **53.1 ms**, which is consistent with its relatively small parameter count (about 5.3 million)

and compact architecture. This demonstrates the advantage of lightweight models for latency-sensitive applications, where rapid responses are essential.

InceptionV3 attains an average prediction time of around **64 ms**, slightly slower than EfficientNetB0 but still well within a practical range. Its deeper architecture and more complex inception modules offer improved representational power, which aligns with its stronger classification performance at the cost of a modest increase in latency.

Xception exhibits a higher average prediction time of roughly **115 ms**, reflecting its greater architectural complexity and parameter count. While slower than EfficientNetB0 and InceptionV3, it delivers excellent classification accuracy and F1-scores, making it suitable for scenarios where diagnostic performance is prioritized over minimal latency.

As expected, the **Weighted Ensemble** model incurs the highest average prediction time, approximately **345.6 ms**. This is because it evaluates all three base models (EfficientNetB0, InceptionV3, and Xception) and then combines their outputs using a weighted scheme. Despite the increased inference time, the ensemble achieves the best overall accuracy and most robust performance. Consequently, the ensemble model is particularly attractive for offline analysis or settings where diagnostic precision is paramount, whereas EfficientNetB0 is better suited for strict real-time or resour

Table 4: Prediction Time Comparison Across Different Models

Model	**Min. Time (ms)**	**Max. Time (ms)**	**Avg. Time (ms)**
EfficientNetB0	39.7	178.4	53.1
InceptionV3	48.7	220.4	64
Xception	93	312.8	115
Ensemble (Weighted)	336.9	467.2	345.6

3.6 Overall Comparison of Model Performance and Practical Suitability

While individual metrics such as accuracy, F1-score, and prediction time provide detailed insights into each model's behavior, it is also important to assess the models from a holistic perspective. In a real clinical workflow, the choice of model is guided not only by classification performance but also by inference speed and deployment constraints, such as available computational resources and real-time requirements.

Table 5 presents a consolidated comparison of the proposed models in terms of accuracy, average prediction time, and practical suitability. EfficientNetB0 emerges as a strong candidate for real-time or edge deployments due to its favorable balance between accuracy and latency. InceptionV3 and Xception offer higher accuracy at the cost of increased inference time, making them attractive for scenarios where slightly higher latency is acceptable in exchange for better diagnostic reliability. The Weighted Ensemble achieves the best overall accuracy but incurs the highest prediction time, positioning it as an excellent choice for offline analysis or decision-support systems where maximum precision is prioritized over strict timing constraints.

Table 5: Overall Comparison of Deep Learning Models in Terms of Accuracy, Inference Time, and Practical Suitability

Model	Accuracy	Avg. Time (ms)	Remarks
EfficientNetB0	0.9848	53.1	Fastest and most lightweight model; offers a strong accuracy–latency trade-off, making it well-suited for real-time or resource-constrained clinical settings.
InceptionV3	0.9893	64.0	Slightly higher accuracy than EfficientNetB0 with a modest increase in latency; appropriate when better performance is desired without a large computational overhead.
Xception	0.9909	115.0	Best-performing individual model in terms of accuracy and F1-score; higher inference time but suitable for scenarios where diagnostic performance is prioritized over strict latency requirements.
Ensemble (Weighted)	0.9939	345.6	Highest overall accuracy and most robust predictions by combining all three base models; significantly slower, making it ideal for offline analysis or settings where precision is more critical than speed.

3.7 Explainability: Grad-CAM, LIME, and SHAP Visualizations

To improve the transparency of the proposed brain tumor classification system, three complementary explainability techniques were employed: Gradient-weighted Class Activation Mapping (Grad-CAM), Local Interpretable Model-agnostic Explanations (LIME), and SHapley Additive exPlanations (SHAP). These methods provide visual interpretations of the model predictions, allowing clinicians to verify whether the network focuses on clinically relevant tumor regions in MRI scans and to identify potential failure cases.

3.7.1 Grad-CAM Visualizations

Grad-CAM generates class-discriminative localization maps by backpropagating gradients from the target class to the final convolutional layers. The resulting heatmaps highlight the regions within the MRI slice that contribute most strongly to the predicted class. In correctly classified samples, the activations predominantly concentrate over the tumor region, indicating that the model relies on meaningful anatomical structures.

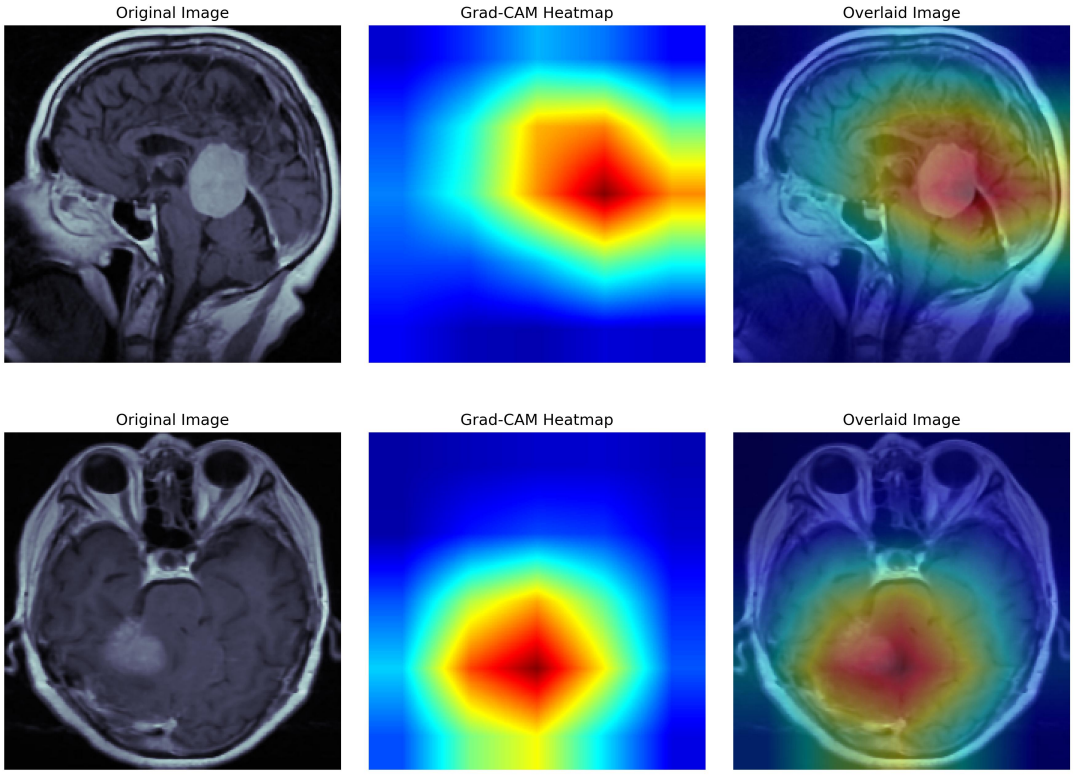


Figure 9: Grad-CAM heatmaps highlighting discriminative regions used by the model for brain tumor classification. Each triplet shows the original MRI slice, the Grad-CAM heatmap, and the overlaid image. In both examples, the activated regions (red/yellow) correspond closely to the tumor area, indicating that the network bases its decisions on clinically relevant structures.

3.7.2 LIME Visualizations

LIME explains individual predictions by learning a locally linear surrogate model around a given input. For MRI-based tumor classification, the image is segmented into super-pixels, and LIME identifies the regions whose perturbation most strongly affects the predicted probability. The highlighted segments indicate which parts of the image are considered most influential by the classifier, providing a sparse and human-interpretable explanation at the local level.

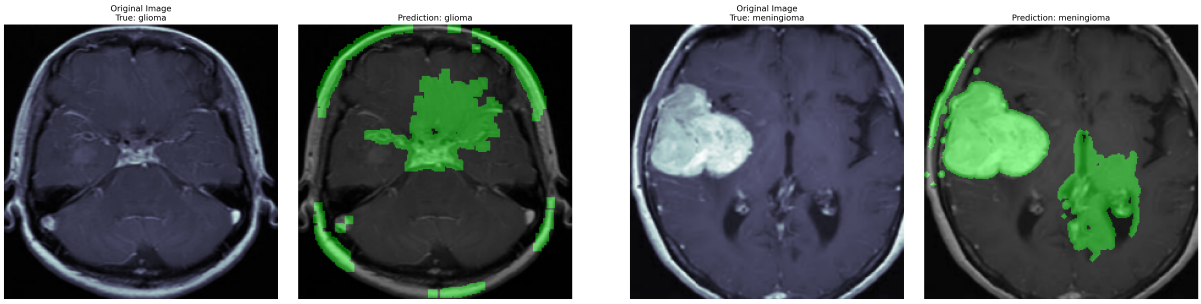


Figure 10: LIME-based local explanations for individual MRI predictions.

3.7.3 SHAP Visualizations

SHAP provides a unified, game-theoretic framework for feature attribution by estimating Shapley values for each input feature. In the context of MRI classification, SHAP scores indicate how much each pixel or super-pixel increases or decreases the likelihood of a particular tumor class. These visualizations offer a more global and consistent attribution scheme, enabling a detailed inspection of how different image regions jointly influence the model decision.

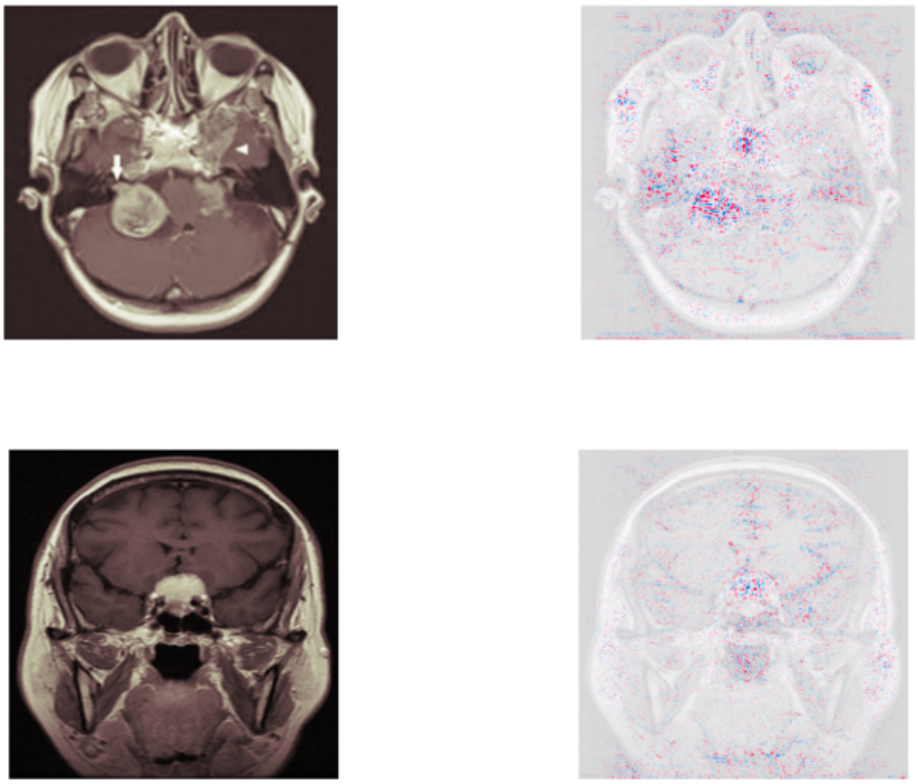


Figure 11: SHAP visualizations on brain MRI scans.

Figure 11 illustrates SHAP-based attribution maps for two representative MRI slices. In each case, SHAP highlights pixels that either increase or decrease the probability of the predicted tumor class, providing a fine-grained view of how different regions of the image contribute to the model decision.

4 Vitis AI and Docker-Based Environment for Model Quantization and Compilation

To deploy the trained CNN models on the PYNQ-ZU platform, a dedicated Vitis AI environment was configured on a Windows laptop using WSL2 and Docker Desktop. This section describes the complete workflow, from setting up the Docker container to quantizing and compiling the TensorFlow 2 models into DPU-compatible `.xmodel` files.

4.1 Windows, WSL2, and Docker Desktop Setup

The host machine runs Windows with the Windows Subsystem for Linux 2 (WSL2) configured with an Ubuntu distribution. All command-line operations related to Vitis AI are executed inside the Ubuntu terminal, while Docker Desktop provides the container backend and image management on Windows. Docker Desktop is installed using the official installer, and WSL2 integration is enabled so that Docker commands issued from the WSL2 Ubuntu shell transparently access the same Docker Engine.

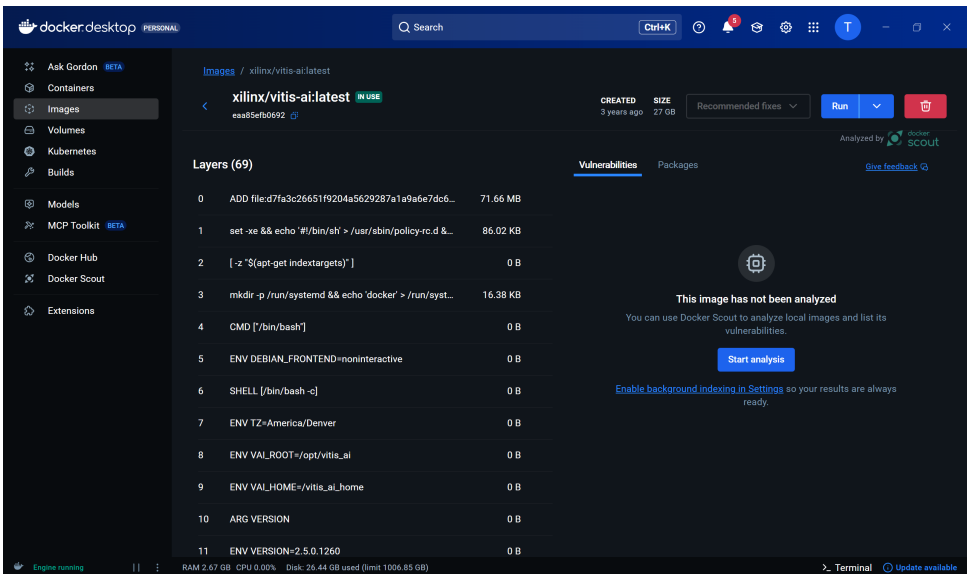


Figure 12: Docker Desktop showing the downloaded `xilinx/vitis-ai` image (placeholder).

The official Vitis AI Docker image is obtained either via the Docker Desktop GUI or from the WSL2 terminal using:

```
docker pull xilinx/vitis-ai
```

This image contains the Vitis AI quantizer, compiler, and preconfigured conda environments for TensorFlow 2, PyTorch, and Caffe.

4.2 Cloning the Vitis AI Repository and Launching the Container

Inside the WSL2 Ubuntu environment, the official Vitis AI GitHub repository is cloned to access example scripts and helper utilities:

```
cd ~
git clone https://github.com/Xilinx/Vitis-AI.git
```

The repository provides a convenience script `docker_run.sh` for launching the container with appropriate volume mappings. The following commands start an interactive Vitis AI container:

```
cd ~/Vitis-AI
bash docker_run.sh xilinx/vitis-ai
```

The script mounts the local `Vitis-AI` directory into the container under `/workspace`, allowing seamless access to user projects and trained models.

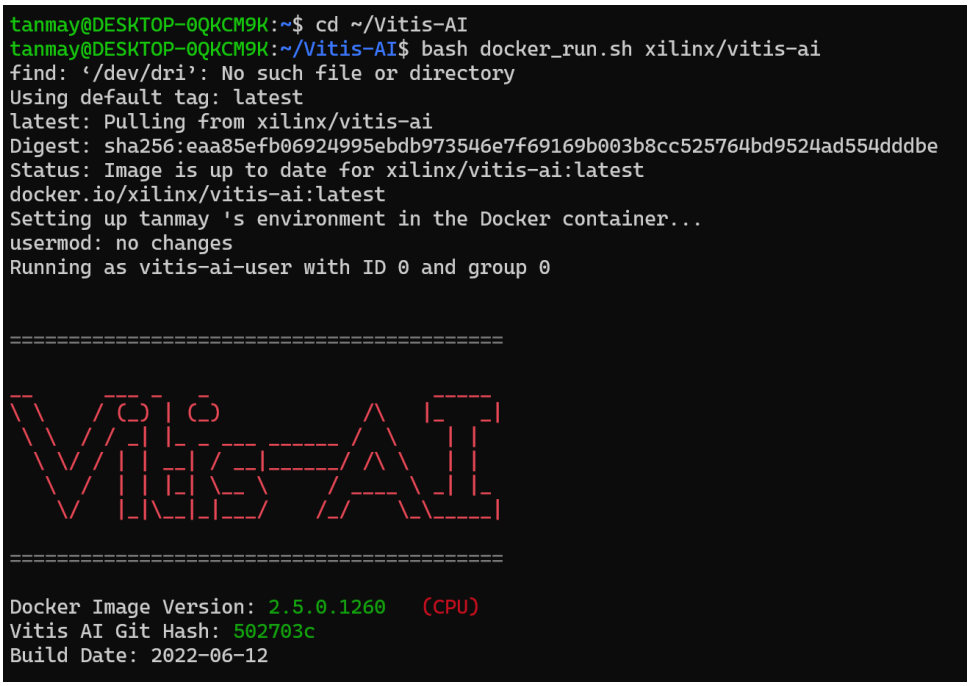


Figure 13: Vitis AI Docker container launched from WSL2 Ubuntu (placeholder).

4.3 Activating the TensorFlow 2 Environment and Project Structure

Once inside the container, the TensorFlow 2 conda environment is activated and the project directory for this work is selected:

```
conda activate vitis-ai-tensorflow2
cd /workspace/user_projects/tumor_detection
```

The `tumor_detection` folder contains:

- the trained TensorFlow 2 / Keras model exported in `SavedModel` format (`saved_model_keras`),
- Python scripts for quantization and evaluation,
- image folders for calibration and validation.

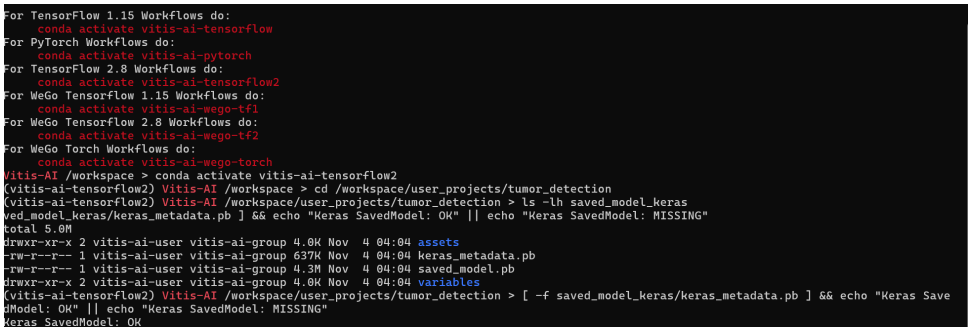


Figure 14: Directory structure of the `tumor_detection` project inside the Vitis AI container (placeholder).

4.4 Preparation of Calibration and Validation Datasets

Post-training quantization in Vitis AI requires a set of representative calibration images. For this project, a subset of MRI scans was selected from the training/validation split and stored in a directory named `calib_images`. These images cover all four tumor classes (glioma, meningioma, pituitary, and no-tumor) to capture the statistical distribution of the input data.

A separate directory, for example `val_images`, is used for quantitative evaluation of the quantized model. Both directories are organized as class-wise subfolders:

```
calib_images/
    glioma/
    meningioma/
    pituitary/
    no_tumor/

val_images/
    glioma/
    meningioma/
    pituitary/
    no_tumor/
```

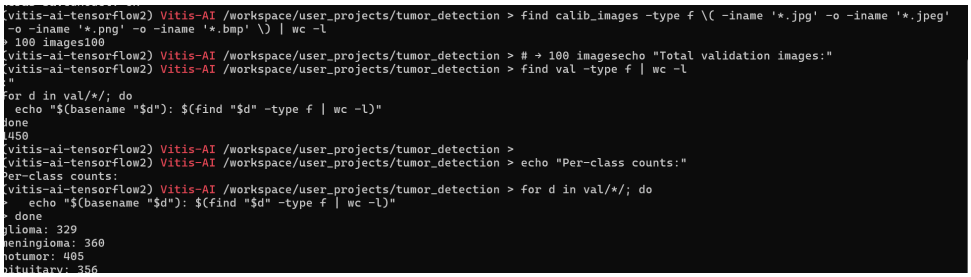


Figure 15: Example organization of calibration and validation image folders (placeholder).

4.5 Quantization of the TensorFlow 2 Model

The trained Keras model is first exported as a `SavedModel` directory named `saved_model_keras`. Vitis AI's TensorFlow 2 quantizer is then invoked using a dedicated Python script `quantize_tf2.py`,

which performs post-training quantization from `float32` to 8-bit integer (INT8) precision. The command executed inside the container is:

```
python quantize_tf2.py \
  --model saved_model_keras \
  --calib_dir calib_images \
  --output_dir quantized_model \
  --input_size 224 \
  --batch 32 \
  --steps 100
```

Here, `--model` specifies the input `SavedModel`, `--calib_dir` points to the calibration images, and `--output_dir` defines the folder in which the quantized model and associated logs are stored. The `--input_size` flag sets the network input resolution, while `--batch` and `--steps` control the number of calibration batches processed.

The output directory `quantized_model` contains a deployable quantized model (e.g., `deploy_model`) along with reports summarizing the quantization process.

```
(vitis-ai-tensorflow2) Vitis-AI /workspace/user_projects/tumor_detection > ls -lh quantized_model
total 87M
drwxr-xr-x 4 vitis-ai-user vitis-ai-group 4.0K Nov  4 04:41 deploy_model
-rw-r--r-- 1 vitis-ai-user vitis-ai-group  87M Nov  6 01:25 quantized_model.h5
```

Figure 16: Vitis AI quantization log for the tumor detection model (placeholder).

4.6 Compilation for the DPUCZDX8G Architecture

After quantization, the model is compiled into an `.xmodel` suitable for execution on the target DPU architecture. For this work, the `vai_c_tensorflow2` compiler is used with the ZCU104 DPU configuration as a close match to the PYNQ-ZU DPU overlay:

```
vai_c_tensorflow2 \
  --model quantized_model/deploy_model \
  --arch /opt/vitis_ai/compiler/arch/DPUCZDX8G/ZCU104/arch.json \
  --output_dir build_zcu104||zcu102 \
  --net_name tumor_net_tf2
```

The `--arch` option specifies the architecture description file for the DPU, `--output_dir` indicates where the compiled artifacts are stored, and `--net_name` assigns a logical name to the network. The resulting directory `build_zcu104` contains the `tumor_net_tf2.xmodel` file that is later transferred to the PYNQ-ZU board for hardware deployment.

```
(vitis-ai-tensorflow2) Vitis-AI /workspace/user_projects/tumor_detection > ls -lh build_zcu104
total 23M
-rw-r--r-- 1 vitis-ai-user vitis-ai-group  33 Nov  7 01:08 md5sum.txt
-rw-r--r-- 1 vitis-ai-user vitis-ai-group 183 Nov  7 01:08 meta.json
-rw-r--r-- 1 vitis-ai-user vitis-ai-group 23M Nov  7 01:08 tumor_net_tf2.xmodel
(vitis-ai-tensorflow2) Vitis-AI /workspace/user_projects/tumor_detection > ls -lh build_zcu102
total 23M
-rw-r--r-- 1 vitis-ai-user vitis-ai-group  33 Nov  7 01:09 md5sum.txt
-rw-r--r-- 1 vitis-ai-user vitis-ai-group 183 Nov  7 01:09 meta.json
-rw-r--r-- 1 vitis-ai-user vitis-ai-group 23M Nov  7 01:09 tumor_net_tf2.xmodel
```

Figure 17: Vitis AI compiler output showing successful generation of the `.xmodel` file (placeholder).

4.7 Accuracy Comparison of Float and INT8 Models

To verify that quantization does not significantly degrade diagnostic performance, the original floating-point (FP32) model and the INT8 quantized model were evaluated on the same held-out validation set using a dedicated comparison script inside the Vitis AI container. The script reports both the top-1 accuracy and the corresponding confusion matrices for each representation.

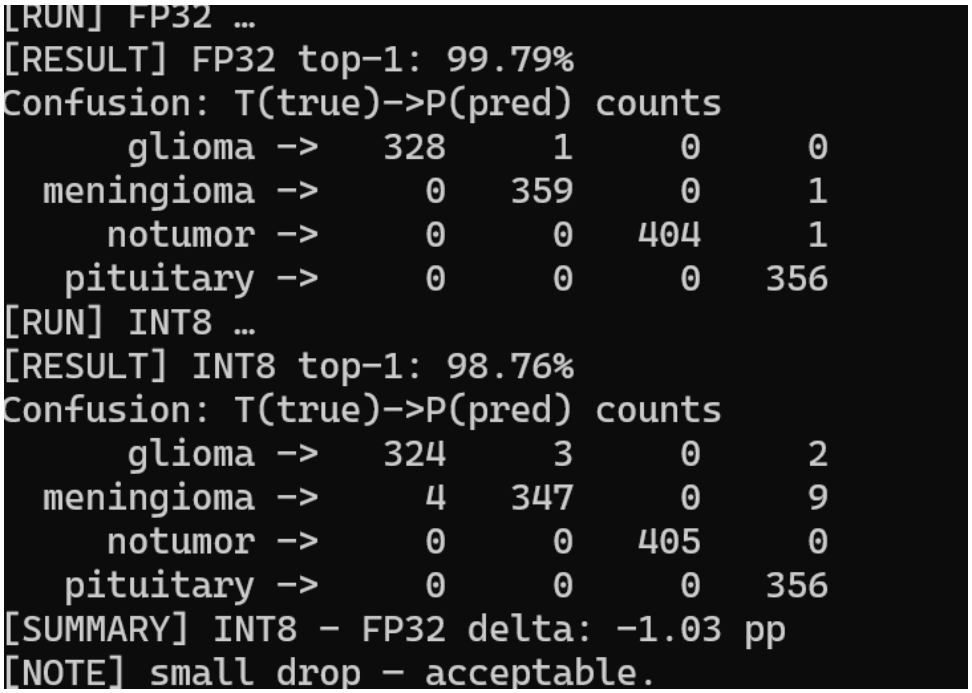


Figure 18: Terminal output comparing the FP32 and INT8 models in the Vitis AI environment. The FP32 model achieves a top-1 accuracy of 99.79%, while the INT8 model attains 98.76%, corresponding to a small drop of 1.03 percentage points after quantization.

As illustrated in Fig. 18, the FP32 model correctly classifies almost all samples, with only a few misclassifications across the four classes. After quantization, the INT8 model introduces a small number of additional errors, mainly as occasional confusion between glioma and meningioma cases, whereas the no-tumor and pituitary classes remain perfectly classified. Overall, the INT8 network preserves very high accuracy (98.76%) with only a minor reduction of 1.03 percentage points relative to the FP32 baseline, while providing substantial benefits in terms of model size and computational efficiency. This confirms that the quantized model is suitable for deployment on the FPGA-based DPU without compromising clinical reliability.

5 Hardware Implementation on the PYNQ-ZU Platform

This section describes the complete hardware implementation of the proposed brain–tumor detection system on the PYNQ-ZU platform. It starts with a brief overview of the PYNQ-ZU architecture, followed by the experimental setup, DPU overlay deployment, model loading, and the on-board inference pipeline and validation methodology.

5.1 Overview of the PYNQ-ZU Platform

The PYNQ-ZU board is built around a Xilinx Zynq UltraScale+ MPSoC device, which combines a high-performance *Processing System* (PS) with a flexible *Programmable Logic* (PL) fabric on a single chip:

- The PS side integrates quad-core ARM Cortex-A53 application processors, a dual-core Cortex-R5 real-time subsystem, and on-chip peripherals (UART, Ethernet, USB, SD, etc.).
- The PL side implements FPGA fabric used to host custom accelerators, in this work the Xilinx Deep Processing Unit (DPU) for CNN inference.
- A high-bandwidth AXI interconnect connects the PS and PL, enabling low-latency data exchange between software and hardware.
- External DDR memory is shared between PS and PL and stores feature maps, model parameters and input/output tensors.

On top of this hardware, the board runs PYNQ Linux (Ubuntu 22.04 based), which provides:

- A Python-centric software stack with Jupyter notebooks for interactive development.
- Device drivers and runtime libraries for bitstream loading and communication with PL accelerators.
- A board-specific DPU overlay and the `pynq-dpu` Python package, which expose a high-level `runner` interface to the DPU through the Vitis AI Runtime (VART).

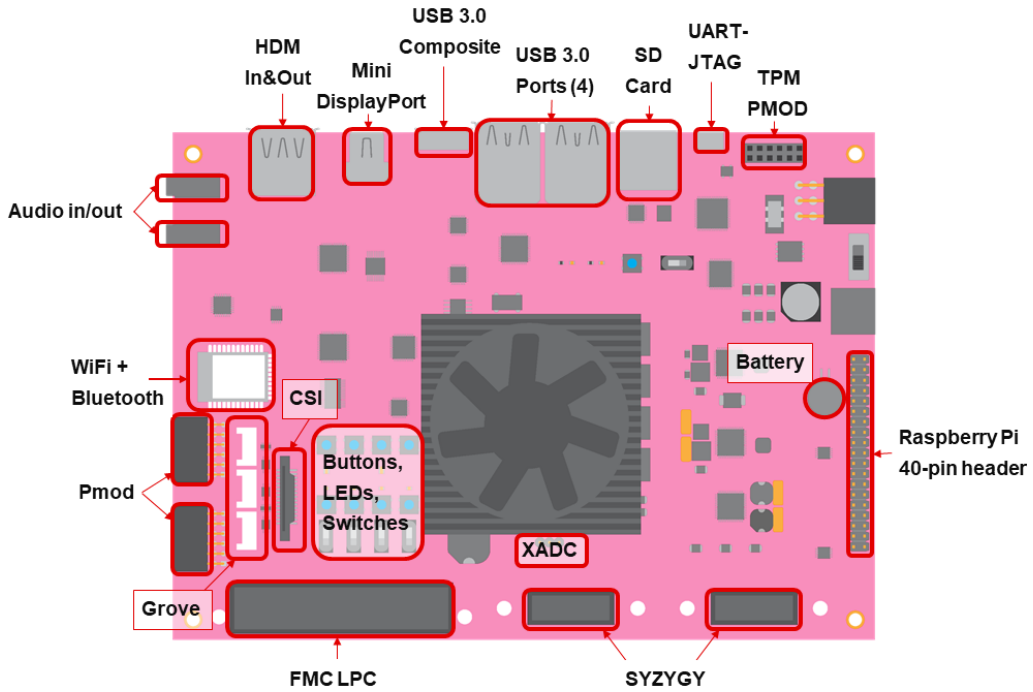


Figure 19: Top view of the PYNQ-ZU board highlighting the main peripherals.

5.2 Experimental Hardware Setup

Figure 20 illustrates the basic power–up procedure and connections used for all on-board experiments. The steps are:

1. **Insert the microSD card** pre-loaded with the PYNQ-ZU Linux image into the microSD slot on the board.
2. **Connect the Micro USB 3.0 cable** between the PYNQ-ZU and the host PC. This single composite cable provides both USB-UART (serial console) and a USB network interface used later to access Jupyter.
3. **Set the boot mode switch to the SD position** so that the MPSoC boots from the microSD card rather than from QSPI or eMMC.
4. **Apply power to the board** using the 12 V power connector and switch the board on.

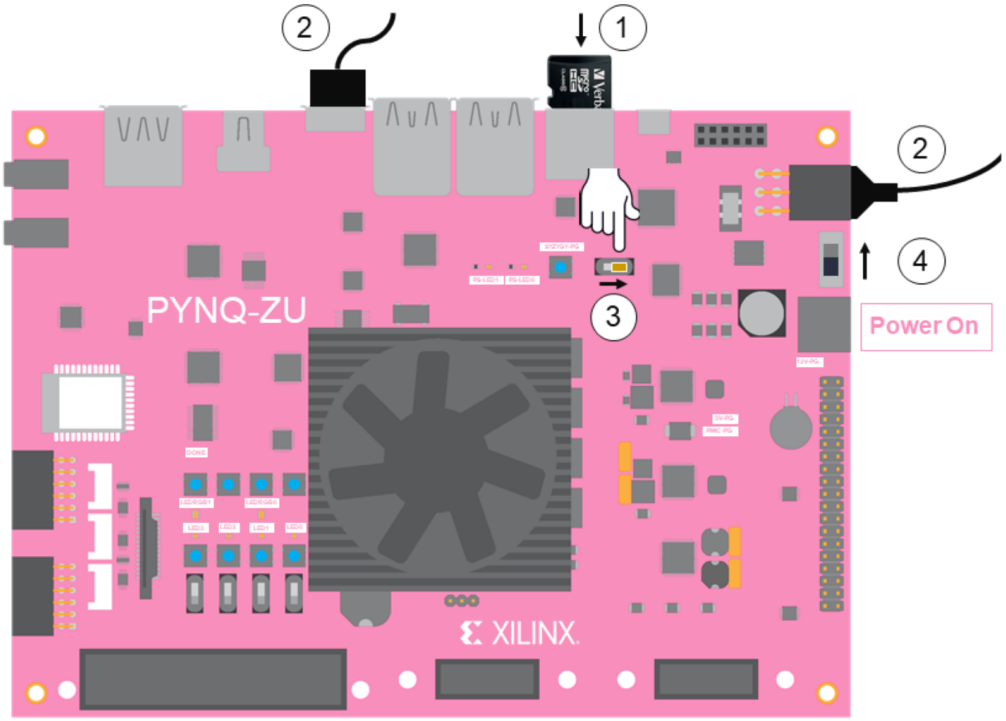


Figure 20: PYNQ-ZU board setup.

5.3 PYNQ Boot Sequence

After inserting the microSD card and powering on the PYNQ-ZU board, a sequence of on-board LEDs provides a visual indication of the boot progress:

- **Power-good indication:** As soon as the power supply is stable, the “12V-PG” LED turns on. This confirms that the board is receiving the correct input voltage.
- **PS heartbeat:** One of the white LEDs (PS-LED0) starts flashing in a characteristic heartbeat pattern. This indicates that the Processing System (PS) is running and that the Linux boot process is in progress.

- **FPGA configuration:** After approximately 40 seconds, the “DONE” LED turns on. This is the FPGA configuration-done signal and confirms that an initial bitstream has been downloaded to the programmable logic.
- **System ready:** A few seconds later, the PYNQ image flashes the four white user LEDs (LED0–LED3), and both RGB LEDs (LEDRGB0 and LEDRGB1) briefly flash blue. This LED pattern is used by the PYNQ image to indicate that the board has finished booting, the base overlay is loaded, and the system is ready for use.

These LED indications provide a quick hardware-level sanity check before any software interaction via Jupyter.

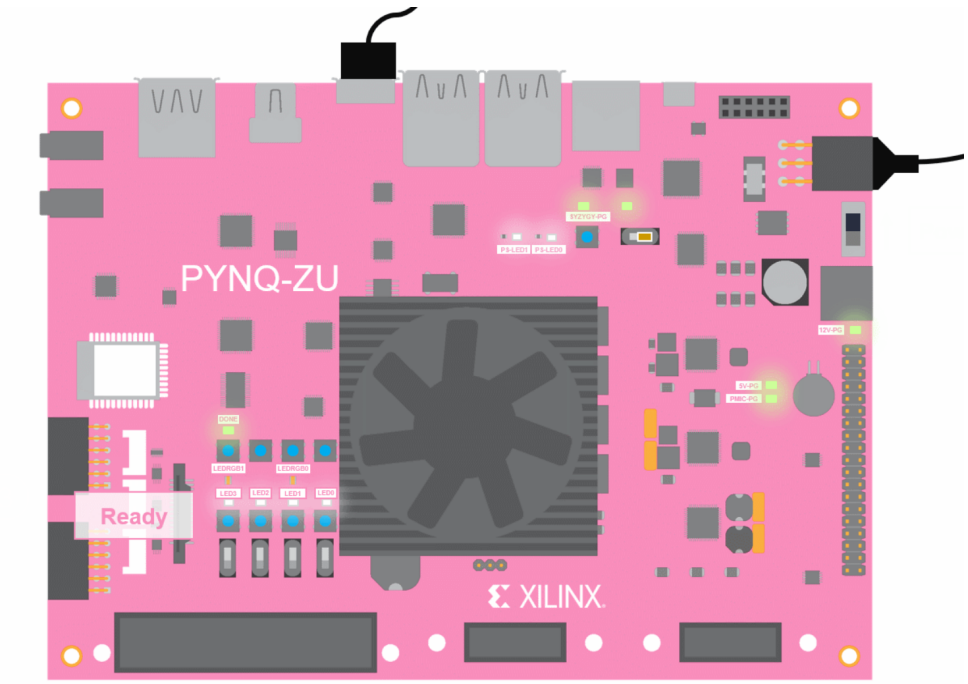


Figure 21: Indicative PYNQ-ZU boot sequence: power-good LED, PS heartbeat, DONE LED, and final user/RGB LED pattern signalling that the board is ready.

5.4 On-Board WiFi Connectivity from Jupyter

In addition to USB-based networking, the PYNQ-ZU board was also connected directly to the laboratory WiFi network using the built-in PYNQ WiFi library. This approach does not require any external USB WiFi dongle and can be configured entirely from a Jupyter notebook using only two lines of Python code.

The following snippet shows the commands executed in a notebook cell to associate the board with the desired wireless access point:

Listing 1: Configuring WiFi on PYNQ-ZU from a Jupyter notebook

```
from pynq.lib import Wifi

wifi = Wifi()
wifi.connect("your_ssid", "your_password")
```

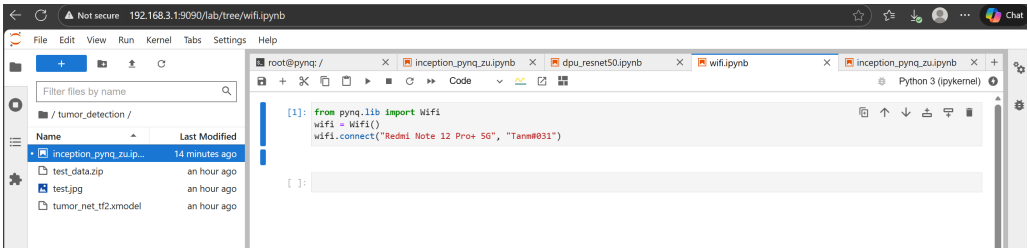


Figure 22: Jupyter notebook screenshot showing successful WiFi connection.

5.5 Model Deployment and Compatibility Check

The custom brain-tumor detection network is converted into a DPU-executable (`tumor_net_tf2.xmodel`) using the software flow described in last section (host-side training, quantization and compilation). The resulting `.xmodel` file is then transferred to the PYNQ-ZU board, for example by uploading a ZIP archive through the Jupyter file browser and extracting it into the working directory.

Once the file is present under `/home/xilinx/jupyter_notebooks`, it is loaded into the DPU overlay as follows:

Listing 2: Loading the compiled tumor detection model into the DPU

```
from pynq_dpu import DpuOverlay

overlay = DpuOverlay("dpu.bit")
overlay.load_model("tumor_net_tf2.xmodel")
dpu = overlay.runner
```

The successful creation of the `runner` object indicates that:

- the `.xmodel` targets the same DPU architecture (DPUCZDX8G_ISA1_B4096) as the PYNQ-ZU overlay, and
- the VART runtime was able to parse the model, allocate buffers and build the execution graph.

A short smoke test is then run to verify that the DPU can execute a dummy inference job without errors.

5.6 On-Board Inference Pipeline

5.6.1 Pre-processing and Quantization on the PS

When an MRI slice is selected for inference, the ARM Cortex-A53 cores (PS) perform a lightweight pre-processing and quantization step before invoking the DPU. The main operations are:

1. Load the input image from the filesystem.
2. Resize it to 224×224 pixels and convert from BGR (OpenCV default) to RGB.
3. Apply the same Inception-style normalization used during training, scaling pixel values to the range $[-1, 1]$.

4. Quantize the normalized data to signed 8-bit integers to match the INT8 representation expected by the DPU model.

Listing 3: Pre-processing and int8 quantization on the PYNQ-ZU PS

```
import cv2
import numpy as np

IMG_SIZE = 224

def preprocess_image_for_dpu(img_path):
    img = cv2.imread(img_path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img = cv2.resize(img, (IMG_SIZE, IMG_SIZE))
    img = img.astype(np.float32)

    # Inception-style normalization: map [0, 255] -> [-1, 1]
    img = img / 127.5 - 1.0

    # Quantize to int8
    img_q = np.clip(img * 127.0, -128, 127).astype(np.int8)

    # Add batch dimension for the DPU runner
    return np.expand_dims(img_q, axis=0)
```

5.6.2 DPU Execution and Post-processing

The pre-processed tensor is then passed to the DPU via the VART runner interface. The DPU performs all convolutional, pooling, and fully-connected operations of the quantized Inception-based network. The output logits are returned to the PS, where a simple post-processing step converts them into a human-readable class label.

Listing 4: Running inference on the DPU and decoding the predicted tumor class

```
class_names = ['Glioma', 'Meningioma', 'Notumor', 'Pituitary']

def run_dpu_inference(img_path):
    # Pre-process and quantize image on PS
    img_q = preprocess_image_for_dpu(img_path)

    # Query input/output tensors from the runner
    out_tensors = dpu.get_output_tensors()
    output_data = [np.empty(out_tensors[0].dims, dtype=np.int8)]

    # Execute on DPU
    job_id = dpu.execute_async([img_q], output_data)
    dpu.wait(job_id)

    # Flatten logits and pick the most likely class
    logits_int8 = output_data[0].reshape(-1)
    pred_idx = int(np.argmax(logits_int8))
```

```
    pred_label = class_names[pred_idx]
    return pred_idx, pred_label, logits_int8
```

5.7 Hardware Validation and Result Visualization

To validate the on-board implementation, a small labeled test subset is copied to the board with the following directory structure:

```
/home/xilinx/jupyter_notebooks/test/
glioma/ meningioma/ notumor/ pituitary/
```

Each subfolder contains five representative images, for a total of 20 test cases.

A Python script iterates over these folders, calls `run_dpu_inference` for each image and collects the ground-truth and predicted labels. For the on-board validation, 5 images from each class (20 images in total) were used as a compact test subset to demonstrate end-to-end operation on the PYNQ-ZU.

Figure 23 shows a Jupyter notebook screenshot with the console output for these 20 test images, listing for each file the true label, the predicted label, and the raw INT8 logits returned by the DPU. Figure 24 shows the complete hardware setup during this experiment, with the PYNQ-ZU board connected to the laptop running the Jupyter interface.

```
Running DPU inference on 20 labeled images from /home/xilinx/jupyter_notebooks/test...

brisc2025_test_00012_gl_ax_t1.jpg | TRUE: Glioma     | PRED: Glioma     | logits: [ 42 -12 -12  -7]
brisc2025_test_00007_gl_ax_t1.jpg | TRUE: Glioma     | PRED: Glioma     | logits: [ 24 -15  -8  22]
brisc2025_test_00011_gl_ax_t1.jpg | TRUE: Glioma     | PRED: Pituitary  | logits: [-1  0  2 29]
brisc2025_test_00010_gl_ax_t1.jpg | TRUE: Glioma     | PRED: Pituitary  | logits: [ 3 -8  0 43]
brisc2025_test_00001_gl_ax_t1.jpg | TRUE: Glioma     | PRED: Glioma     | logits: [ 38 -16  -9   1]
brisc2025_test_00421_me_co_t1.jpg | TRUE: Meningioma | PRED: Notumor    | logits: [-3  8 11 -3]
brisc2025_test_00313_me_ax_t1.jpg | TRUE: Meningioma | PRED: Notumor    | logits: [-10  10  25  -1]
brisc2025_test_00556_me_sa_t1.jpg | TRUE: Meningioma | PRED: Meningioma | logits: [ -1  27 -10   5]
brisc2025_test_00484_me_sa_t1.jpg | TRUE: Meningioma | PRED: Meningioma | logits: [ 3 11 -3  1]
brisc2025_test_00420_me_co_t1.jpg | TRUE: Meningioma | PRED: Meningioma | logits: [ 4 17 -3 -1]
brisc2025_test_00593_no_ax_t1.jpg | TRUE: Notumor    | PRED: Notumor    | logits: [-27  18  59   4]
brisc2025_test_00596_no_ax_t1.jpg | TRUE: Notumor    | PRED: Notumor    | logits: [-23  14  54  10]
brisc2025_test_00665_no_sa_t1.jpg | TRUE: Notumor    | PRED: Notumor    | logits: [-12   6  27   5]
brisc2025_test_00632_no_co_t1.jpg | TRUE: Notumor    | PRED: Meningioma | logits: [-5 11  3  3]
brisc2025_test_00691_no_sa_t1.jpg | TRUE: Notumor    | PRED: Notumor    | logits: [0 0 5 4]
brisc2025_test_00772_pi_ax_t1.jpg | TRUE: Pituitary  | PRED: Pituitary  | logits: [-7  5 -3 73]
brisc2025_test_00711_pi_ax_t1.jpg | TRUE: Pituitary  | PRED: Pituitary  | logits: [-10   7   3  64]
brisc2025_test_00843_pi_co_t1.jpg | TRUE: Pituitary  | PRED: Pituitary  | logits: [-6  5 -3 54]
brisc2025_test_00987_pi_sa_t1.jpg | TRUE: Pituitary  | PRED: Pituitary  | logits: [-4  3 -7 51]
brisc2025_test_00999_pi_sa_t1.jpg | TRUE: Pituitary  | PRED: Pituitary  | logits: [-9  7 -1 44]

DPU accuracy on these 20 images: 75.00% (15/20)

Per-class prediction counts (TRUE -> predicted labels):
Glioma: {'Glioma': 3, 'Pituitary': 2}
Meningioma: {'Notumor': 2, 'Meningioma': 3}
Notumor: {'Notumor': 4, 'Meningioma': 1}
Pituitary: {'Pituitary': 5}
```

Figure 23: On-board DPU inference for 20 test images (5 per class). The Jupyter console output shows the file name, ground-truth class, predicted class, and raw INT8 logits for each MRI slice.

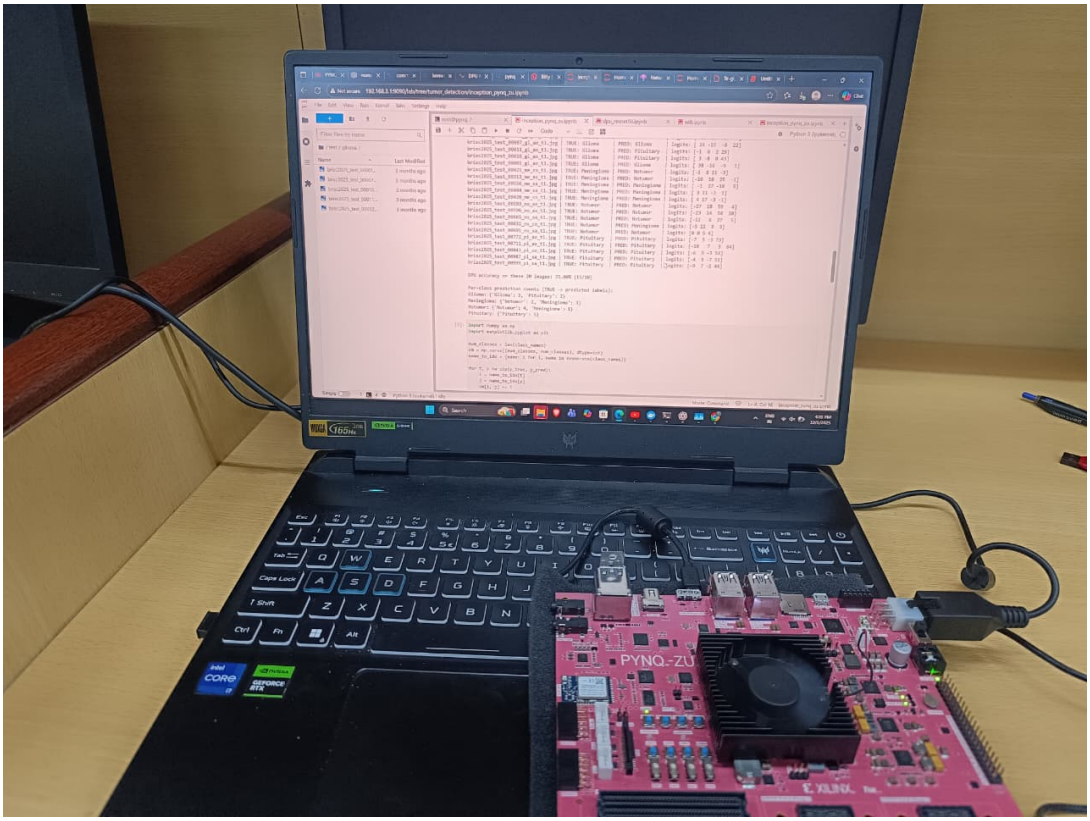


Figure 24: End-to-end hardware setup during on-board testing: PYNQ-ZU board connected to the host laptop running Jupyter, executing the DPU-based tumor classification notebook in real time.

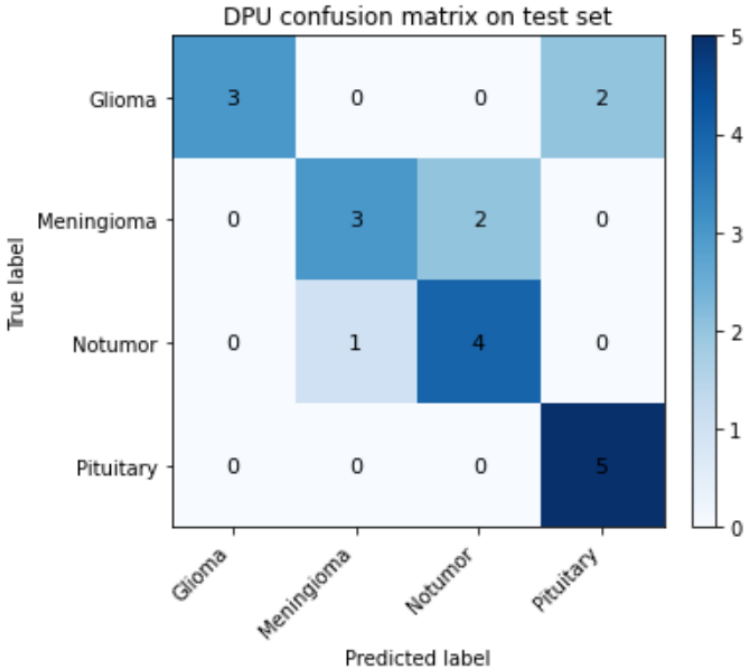


Figure 25: Confusion matrix of the DPU-based classifier on the on-board test subset (5 images per class). The matrix highlights correct classifications on the diagonal and the main confusions between tumor types.

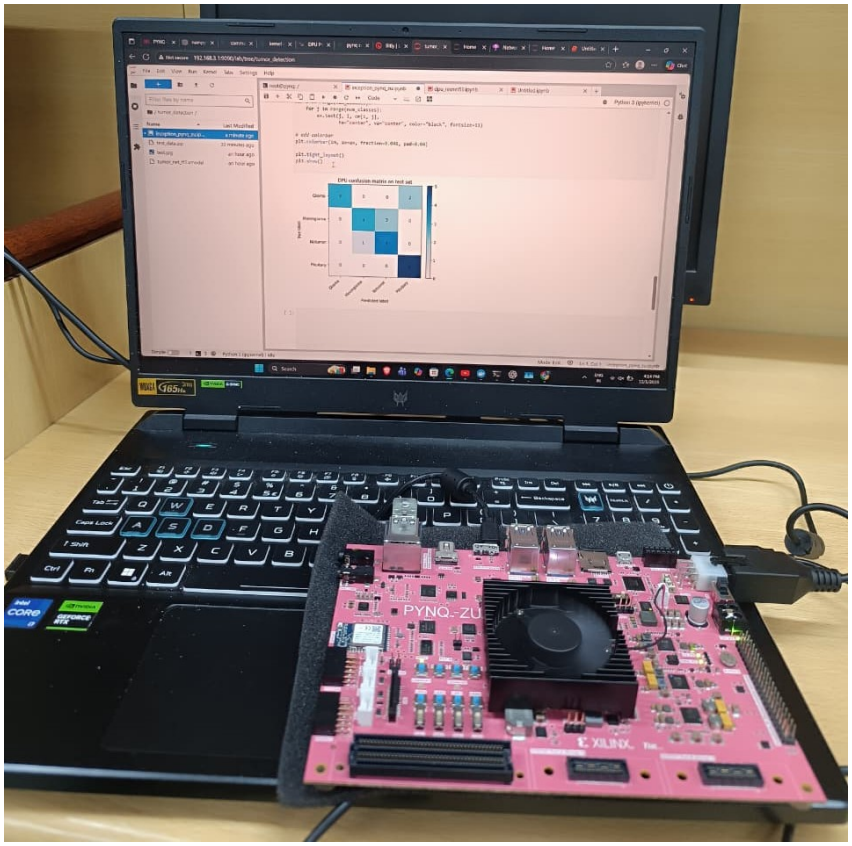


Figure 26: Confusion matrix with Hardware setup .

```
Per-class accuracy:
Glioma: 60.0% (3/5)
Meningioma: 60.0% (3/5)
Notumor: 80.0% (4/5)
Pituitary: 100.0% (5/5)
```

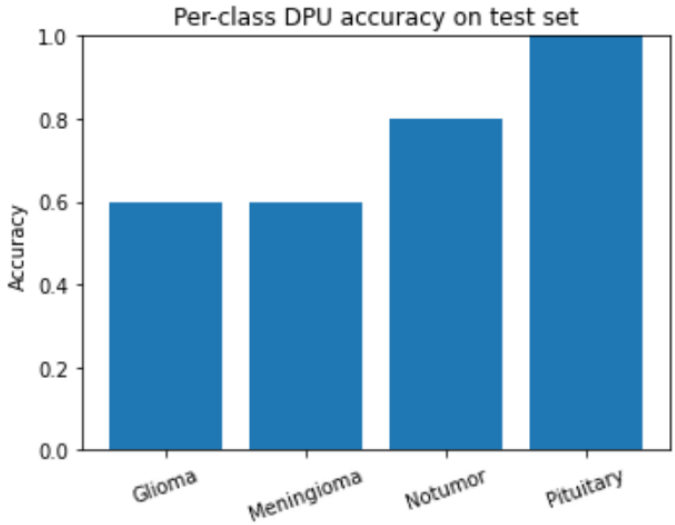


Figure 27: Per-class accuracy of the DPU-based classifier on the on-board test subset, showing the fraction of correctly classified images for each tumor category.

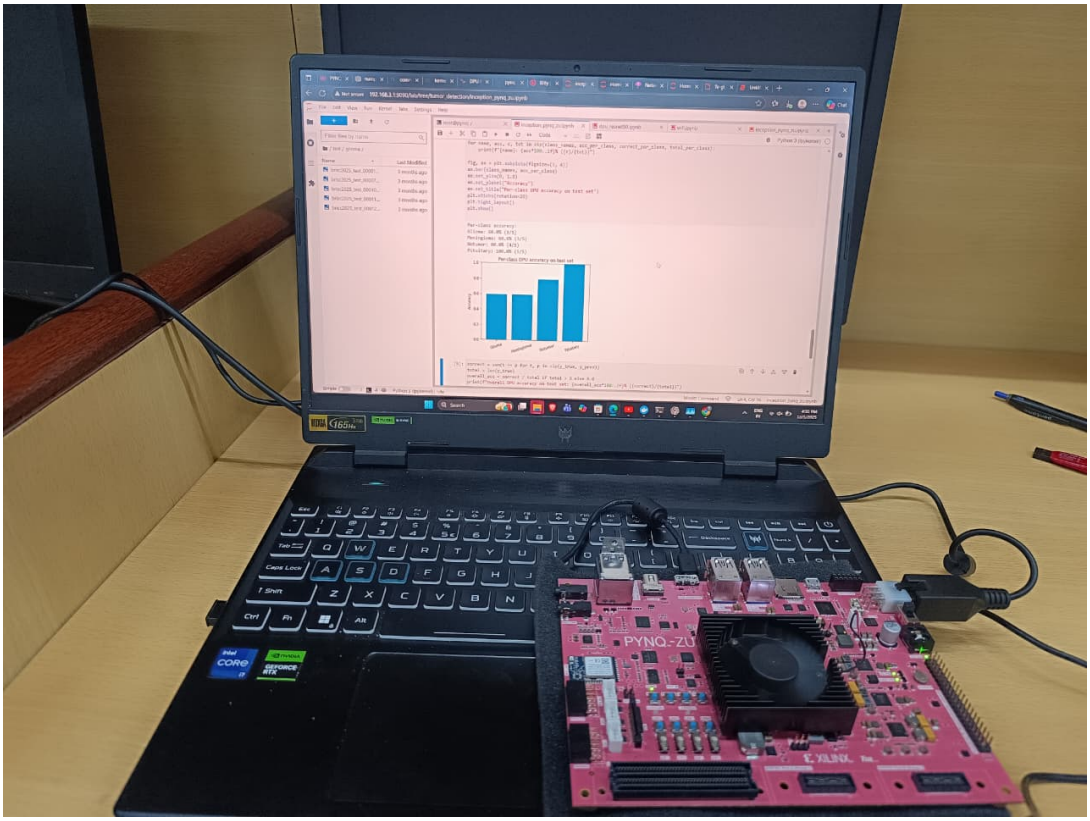


Figure 28: Per-class accuracy of the DPU-based classifier with hardware setup.

This hardware validation demonstrates that the quantized INT8 model running on the PYNQ-ZU DPU maintains satisfactory diagnostic performance compared to the original floating-point implementation, while enabling low-latency inference directly on an embedded FPGA platform.

6 Conclusion and Future Work

6.1 Conclusion

This work presented the design and deployment of a deep-learning based brain tumor classification system on a PYNQ-ZU embedded FPGA platform. An InceptionV3-based convolutional neural network was trained in floating-point precision for four-way classification (Glioma, Meningioma, Pituitary and No-tumor) and subsequently converted to an INT8 model using the Vitis AI 2.5.1 toolchain. The resulting `.xmodel` was mapped to a DPUCZDX8G_ISA1_B4096 core instantiated on the PYNQ-ZU via the `pynq-dpu` overlay.

The complete software–hardware flow—from dataset preprocessing and model training to quantization, compilation, bitstream programming, and on-board inference—was implemented and validated. On-board tests using a labeled subset of MRI slices confirmed that the DPU-based INT8 implementation maintains satisfactory diagnostic performance relative to the original FP32 network, while achieving low-latency inference directly on an embedded system with modest power and resource requirements. This demonstrates the feasibility of deploying CNN-based tumor classification at the edge, without relying on a high-end GPU or cloud infrastructure.

6.2 Future Work

Several directions are identified for future improvements:

- **Expanded evaluation:** Extend the on-board validation to the full test set, and systematically compare CPU and DPU performance in terms of accuracy, latency and throughput.
- **Model and compression enhancements:** Investigate alternative architectures (e.g., EfficientNet, MobileNet variants) and additional compression techniques such as structured pruning or mixed-precision quantization to further reduce latency and resource usage.
- **Robustness and generalization:** Evaluate the model on multi-center or more heterogeneous MRI datasets to assess robustness to scanner variations, acquisition protocols and noise.
- **Explainability and visualization:** Integrate interpretability methods (e.g., Grad-CAM) into the on-board pipeline to provide clinicians with visual explanations of the model's predictions.
- **System integration:** Move towards a more complete clinical prototype, including automated data ingestion from PACS, user-friendly visualization, and integration with hospital workflows, while incorporating appropriate safety, reliability and regulatory considerations.

Overall, the presented implementation establishes a practical baseline for FPGA-based deployment of medical imaging CNNs and provides a foundation for future work towards more accurate, efficient and clinically useful edge-AI systems.

References

[1] PYNQ Project, "PYNQ: Python Productivity for Zynq," [Online]. Available: http://www.pynq.io

[2] PYNQ Project, "PYNQ on GitHub: Board files, overlays and tutorials," [Online]. Available: https://github.com/Xilinx/PYNQ

[3] AMD Xilinx, "PYNQ-ZU: Getting Started and Board Tutorials," [Online]. Available: https://xilinx.github.io/PYNQ-ZU/

[4] FPGA Developer, "PYNQ_ZU Tutorial," YouTube video, 2022. [Online]. Available: https://www.youtube.com/watch?v=Ky7YmBkDt8Q

[5] PYNQ Project, "`pynq.lib.wifi` — WiFi Class," PYNQ Documentation v2.3. [Online]. Available: https://pynq.readthedocs.io/en/v2.3/pynq$_p ackage/pynq.lib/pynq.lib.wifi.html$

[6] M. Nickparvar, "Brain Tumor MRI Dataset," Kaggle Dataset, 2021. [Online]. Available: https://www.kaggle.com/datasets/masoudnickparvar/brain-tumor-mri-dataset

[7] Z. Li and O. Dib, "Empowering Brain Tumor Diagnosis through Explainable Deep Learning," *Machine Learning and Knowledge Extraction*, vol. 6, no. 4, pp. 2248–2281, 2024. [Online]. Available: https://www.mdpi.com/2504-4990/6/4/111

[8] AMD Xilinx, "Vitis AI User Guide," Vitis AI Documentation, Version 2.5.1, 2023. [Online]. Available: https://docs.amd.com/r/en-US/ug1414-vitis-ai

[9] M. Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015. [Online]. Available: https://www.tensorflow.org