ACKNOWLEDGEMENT

It is often said that we stand on the shoulders of those who came before us, and this project is a testament to that. We owe our success to the valuable contributions and support of many individuals.

We would like to express our sincere gratitude to **Prof. Shrihari S Pujar** for her constant guidance and encouragement throughout the project. Her expertise and unwavering support were crucial in helping us complete this project with great interest and dedication.

We also take this opportunity to extend our deepest thanks to **Dr. V.S. Malemath**, Professor and HOD, for his valuable guidance and continuous encouragement, which motivated us to work harder and stay focused.

Our heartfelt appreciation goes to **Dr. S.F. Patil**, Principal, for his blessings and for providing an environment conducive to learning and growth.

We are extremely grateful to **KLE Technological University** and **Dr. M.S. Sheshgiri College of Engineering and Technology**, Belagavi, for giving us the opportunity to pursue this project and for being instrumental in shaping our academic and professional futures.

Finally, we would like to acknowledge all the individuals who have directly or indirectly contributed to the completion of this project. Your valuable suggestions and support have been greatly appreciated.

Project Members:

1. Prajwal Hiremath
2. Harish Baragali
3. Shubham Swami
4. Shridhar Patil

1. INTRODUCTION

1.1 PROJECT DESCRIPTION

Stroke is one of the most serious health issues today, often leading to long-term disability or even death if not detected early. The faster a stroke is diagnosed, the better the chances are for effective treatment and recovery. However, traditional methods like physical exams and brain scans can sometimes take time and may depend heavily on the doctor's judgment.

To help with this, we decided to build a machine-learning model that can detect strokes more quickly and accurately. Our project uses deep learning techniques, with a special focus on the ReLU (Rectified Linear Unit) activation function. ReLU is commonly used in neural networks because it's fast, simple, and helps the model learn patterns more effectively, especially in large and complex medical data.

The goal of our project is to create a smart system that can support doctors by analyzing patient data and predicting the possibility of a stroke. In this paper, we explain how we designed and trained our model, how it works, and how well it performs when tested on real data

1.2 OBJECTIVE OF PROJECT

- **To provide doctors with good software to identify strokes and their causes:** Develop an AI-based tool that helps doctors accurately detect strokes (hemorrhagic or ischemic) from CT/MRI scans, improving diagnostic confidence.
- **Save patient's time.**: To assess the performance of various machine learning algorithms (e.g., Logistic Regression, Decision Trees, Random Forest, SVM) and determine the most effective one for this application.
- **Provide a solution appropriately at early stages**: To design a user-friendly interface that allows healthcare professionals to input patient data easily and receive predictions in real time.
- **Get timely consultation**: Assist healthcare professionals in making quicker decisions, leading to prompt consultations and better patient outcomes.

2. LITERATURE SURVEY

2.1 EXISTING AND PROPOSED SYSTEM

EXISTING SYSTEM

Overview of Existing Systems

The early detection of brain strokes traditionally relies on methods such as clinical examination, MRI (Magnetic Resonance Imaging), and CT (Computed Tomography) scans, interpreted by experienced radiologists. While effective, these methods are often time-consuming, subjective, and dependent on human expertise. In emergency situations, this delay can significantly affect treatment outcomes and patient recovery.

Limitations of Existing Systems

1. **Dependence on Radiologists**: Traditional systems rely heavily on expert radiologists for interpreting MRI and CT images. This introduces subjectivity and may lead to inconsistent or delayed diagnosis, especially in under-resourced settings.

2. **Time-Consuming Processes**: Manual image analysis and reporting take valuable time during the critical early stages of a stroke, where every minute can affect the patient's recovery outcome ("time is brain").

3. **Limited Generalization**: Many deep learning models are trained on small or homogeneous datasets. As a result, their performance may degrade when applied to new patients or images from different sources or imaging devices.

4. **Hardware and Infrastructure Requirements**: Advanced imaging tools like MRI and high-resolution CT scanners are expensive and often unavailable in rural or emergency environments.

5. **Lack of Real-Time Integration**: Most existing models are not embedded in real-time clinical workflows or portable devices, limiting their application in time-sensitive or mobile scenarios.

PROPOSED SYSTEM

Overview of the Proposed System

The proposed system is a deep learning-based framework designed to detect brain strokes automatically from MRI and CT imaging data. By leveraging Convolutional Neural Networks (CNNs) and the Rectified Linear Unit (ReLU) activation function, the system aims to improve the speed and accuracy of stroke detection, reducing reliance on manual interpretation.

Key Features of the Proposed System

1. **Data Dependency**: The model's performance is heavily dependent on the quality and diversity of the training data. Biases in the dataset may affect real-world applicability.

2. **Limited Interpretability**: Like most deep learning models, the system operates as a "black box," providing little interpretability or rationale behind its predictions, which can be a challenge in clinical environments.

3. **Hardware Constraints**: Although optimized, the model still requires a reasonably powerful system (preferably with GPU) to ensure fast and smooth performance, which might not be available in all settings.

4. **Generalization Challenges**: The model may not generalize well to images from different machines, hospitals, or patient demographics not represented in the training data.

5. **No Multi-modal Integration**: The current version does not combine imaging data with clinical parameters (e.g., patient history, symptoms), which could further improve diagnostic accuracy.

2.2 FEASIBILITY STUDY

The use of the ReLU (Rectified Linear Unit) activation function in this stroke detection system is both practical and effective. Technically, ReLU is simple, avoids vanishing gradients, and enables faster training of deep CNNs. Operationally, it allows real-time predictions with low resource usage, making the system suitable for clinical and portable applications. Economically, ReLU is free to use, reduces training time, and lowers computational costs. Although ReLU can suffer from inactive neurons ("dying ReLU"), this is manageable with proper initialization or using alternatives like Leaky ReLU. Overall, ReLU is a feasible and efficient choice for this deep learning model.

1. TECHNICAL FEASIBILITY

The technical feasibility of the proposed brain stroke detection system is well supported by current advancements in machine learning, medical imaging, and software development. The system is based on the integration of Convolutional Neural Networks (CNNs), a proven architecture for image analysis tasks:

- **Availability of Datasets**: Multiple publicly available and well-annotated datasets for brain MRI and CT scans are accessible through platforms like Kaggle and The Cancer Imaging Archive (TCIA). These datasets provide a reliable foundation for model training and validation.
- **Deep Learning Frameworks**: The system utilizes Python along with TensorFlow and Keras, both of which are mature and widely adopted deep learning libraries that support efficient model design, training, and deployment.
- **Processing Capability**: With GPU acceleration support in TensorFlow, the training process is significantly faster and can be executed on modern consumer-grade hardware or cloud platforms, reducing the need for high-end infrastructure.
- **Software Environment**: The development and testing environment is managed using Anaconda, which simplifies package management and ensures reproducibility across different systems.
- **Model Performance**: The CNN architecture, enhanced with ReLU activation functions and training techniques such as data augmentation and early stopping, shows high accuracy and generalization capability on test data, validating its technical soundness.

2. OPERATIONAL FEASIBILITY

The proposed brain stroke detection system demonstrates strong operational feasibility, as it aligns well with practical use cases in healthcare and is designed to function effectively in clinical and non-clinical environments.:

- **Ease of Use**: The system features a user-friendly graphical interface built with Tkinter, enabling users (e.g., doctors, and technicians) to upload brain scan images, receive predictions, and visualize results without requiring technical expertise.
- **Real-Time Predictions**: The model is optimized for fast inference, allowing near-instantaneous predictions. This is crucial for emergency diagnosis where time-sensitive decision-making is critical.
- **Minimal Training Required**: Medical staff can operate the system with minimal training, as the interface is intuitive and requires only basic interaction like image upload and result interpretation.
- **Portability and Integration**: The desktop-based application can be deployed in various healthcare settings, including hospitals, diagnostic labs, and even in mobile health units, provided a basic computing setup is available.
- **Result Logging**: The system automatically logs each prediction along with the image name, confidence score, and timestamp in a CSV file. This feature enhances traceability and can be valuable for audits, research, and medical record-keeping.
- **Scalability**: As a modular system developed using Python and TensorFlow, it can be extended in the future to include additional features like multi-stroke classification, integration with electronic health records (EHR), or deployment on web/mobile platforms.
- **Support and Maintenance**: Since it is developed using open-source tools, the system benefits from community support and regular updates, making maintenance and troubleshooting more manageable

3. ECONOMIC FEASIBILITY

when compared to traditional diagnostic enhancements or commercial AI-based medical software. The system leverages open-source tools and publicly available datasets, minimizing development and deployment costs while maximizing accessibility and scalability.

- **No Licensing Costs**: The system is developed using open-source frameworks such as Python, TensorFlow, Keras, and Tkinter. These tools are free to use, eliminating the need for costly software licenses or proprietary development environments.
- **Free Datasets**: Public datasets sourced from platforms like Kaggle and The Cancer Imaging Archive (TCIA) significantly reduce the cost and time required for data acquisition.
- **Reduced Operational Costs**: By enabling early and automated stroke detection, the system may help reduce diagnostic errors and unnecessary testing, thus lowering long-term treatment and healthcare costs.
- **Minimal Training and Support Costs**: The intuitive interface minimizes the need for extensive training programs or specialized operators, reducing the overhead typically associated with new medical technologies

2.3 TOOLS AND TECHNOLOGIES USED

The following tools and technologies were utilized in the development of the Brain Stroke Detection System:

Programming Language: Python was selected for its extensive libraries and community support in data science and machine learning. Its simplicity and readability make it an ideal choice for rapid development and prototyping.

Machine Learning Libraries:

Scikit-learn: Scikit-learn is a powerful Python library for classical machine learning algorithms and evaluation metrics. While not explicitly mentioned in the document, it's likely used for computing.

Keras: Keras is a high-level neural networks API that runs on top of TensorFlow. It simplifies building and training deep learning models by providing easy-to-use functions for layers, optimizers, loss functions, and evaluation metrics.

2.4 ALGORITHMS

LOGISTIC REGRESSION

Logistic Regression is a **supervised learning algorithm** used for **binary classification** problems. It predicts the probability that a given input belongs to one of two classes (e.g., stroke or no stroke). Unlike linear regression, which outputs a continuous value, logistic regression uses a **sigmoid function** to produce a probability between 0 and 1.

Logistic Regression Assumptions:

- **Output**: Probability score mapped to a class (e.g., 1 = stroke, 0 = no stroke).
- **Sigmoid Function**: σ(x)=11+e−x\sigma(x) = \frac {1} {1 + e^{-x}} σ(x)=1+e−x1, compresses the input to a [0, 1] range.
- **Decision Boundary**: A threshold (usually 0.5) is set to classify predictions into classes.

Use in Stroke Detection

Although this project primarily uses CNNs for image-based classification, **logistic regression** could be applied in simpler stroke prediction models based on:

- Patient attributes (age, blood pressure, cholesterol, smoking status, etc.)
- Lab results or vital signs.

3. HARDWARE AND SOFTWARE REQUIREMENTS

HARDWARE REQUIREMENTS:

- **Computer**: 16 GB RAM, quad-core processor
- **Graphics Card**: 2 GB video memory
- **Storage**: 256 GB SSD
- **Processor**: Intel Core i5 or above.
- **Hard disk:** 10 GB of available space or more.

SOFTWARE REQUIREMENTS:

- **Languages**: Python
- **Frameworks**: TensorFlow, Keras.
- **Image Processing**: OpenCV
- **Data Tools**: NumPy, Pandas
- **Visualization**: Matplotlib
- **Environments**: Anaconda, VS Code.
- **Medical Imaging**: SimpleITK,
- **Datasets:** Kaggle.
- **OS:** Windows

3.1 FUNCTIONAL REQUIREMENTS

Image Upload Interface: Users can upload MRI or CT scan images for stroke analysis.

Preprocessing Pipeline: Resizes, normalizes, and enhances the input image using filters and contrast adjustments.

Stroke Detection via CNN: The model classifies the image as either "stroke" or "no stroke" using a trained Convolutional Neural Network with ReLU activation.

Result Logging: Logs predictions with filename, result, confidence, and timestamp in a CSV file for later reference.

3.2 NON-FUNCTIONAL REQUIREMENTS

Performance: The system should provide a prediction with high accuracy (validation accuracy ≥ 99%) and respond within a few seconds.

Usability: Simple, user-friendly interface for medical professionals with minimal training.

Security: Basic protection for local file access and result logging to prevent unauthorized modifications.

Maintainability: Modular code structure using TensorFlow and Keras to support easy updates or model retraining.

Portability: Platform-independent application; compatible with Windows, Linux, and MacOS using Python and Anaconda

3.3 MODULES

The Brain Stroke Detection System is structured into several key modules, each responsible for distinct functionalities. The main modules include:

1. **Data Acquisition Module**
2. **Image Preprocessing Module**
3. **Model Architecture Module**
4. **Training and Evaluation Module**
5. **Prediction and Inference Module**
6. **Graphical User Interface (GUI) Module**
7. **Logging and Report Module**

1. Data Acquisition Module

Overview: Collects MRI and CT brain scan images from publicly available datasets or clinical sources.

Key Functions:

- Load datasets from local or online sources (e.g., Kaggle, PhysioNet).
- Label data for classification (stroke vs. no stroke).
- Organize data into training, validation, and test sets.

2. Image Preprocessing Module

Overview: Prepares raw medical images for input into the machine learning model.

Key Functions:

- Resize images to uniform dimensions (e.g., 224x224).
- Normalize pixel values.
- Apply noise-reduction filters (e.g., Gaussian Blur).
- Enhance contrast using histogram equalization.

3. Model Architecture Module

Overview: Defines the deep learning model structure using CNN and ReLU.

Key Functions:

- Construct CNN layers: Convolution, MaxPooling, Fully Connected.
- Use ReLU activation to introduce non-linearity.
- Final output layer with Softmax for binary classification.

4 Training And Evaluation Module

Overview: Trains the CNN model and evaluates its performance.

Key Functions:

- Compile model with loss function (categorical cross-entropy) and optimizer (Adam).
- Train model with early stopping and learning rate scheduling.
- Evaluate accuracy, precision, recall, F1-score, and display confusion matrix.

5 Prediction And Inference Module

Overview: Applies the trained model to new input images for live stroke detection.

Key Functions:

- Load-trained model weights.
- Accept new image input from the user.
- Predict stroke or no stroke and return confidence score.

6 Graphical User Interface (Gui) Module

Overview: Provides a user-friendly desktop application using Tkinter

Key Functions:

- Allow users to upload MRI/CT images.
- Display predictions, images, and confidence scores.
- Theme support with light/dark mode and animation effects.

7 Logging and Report Module

Overview: Maintains a log of predictions for future analysis and audits.

Key Functions:

- Save results to a CSV file with a timestamp, image name, predicted class, and confidence score.
- Export basic reports for clinical use.

4. SYSTEM DESIGN

4.1 SYSTEM PERSPECTIVE

The proposed stroke detection system is designed as a decision-support tool for clinicians and medical practitioners. It does not replace traditional diagnostics but enhances them by providing fast, AI-based analysis of brain CT and MRI scans.

Standalone Application: The system operates as a desktop application using a local deep learning model, requiring no internet connectivity.

User Interaction: Designed for radiologists, medical researchers, and healthcare providers, allowing them to upload brain scans and receive instant predictions.

Integration Potential: This can be integrated into telemedicine kits or mobile stroke units, aiding in pre-hospital diagnosis.

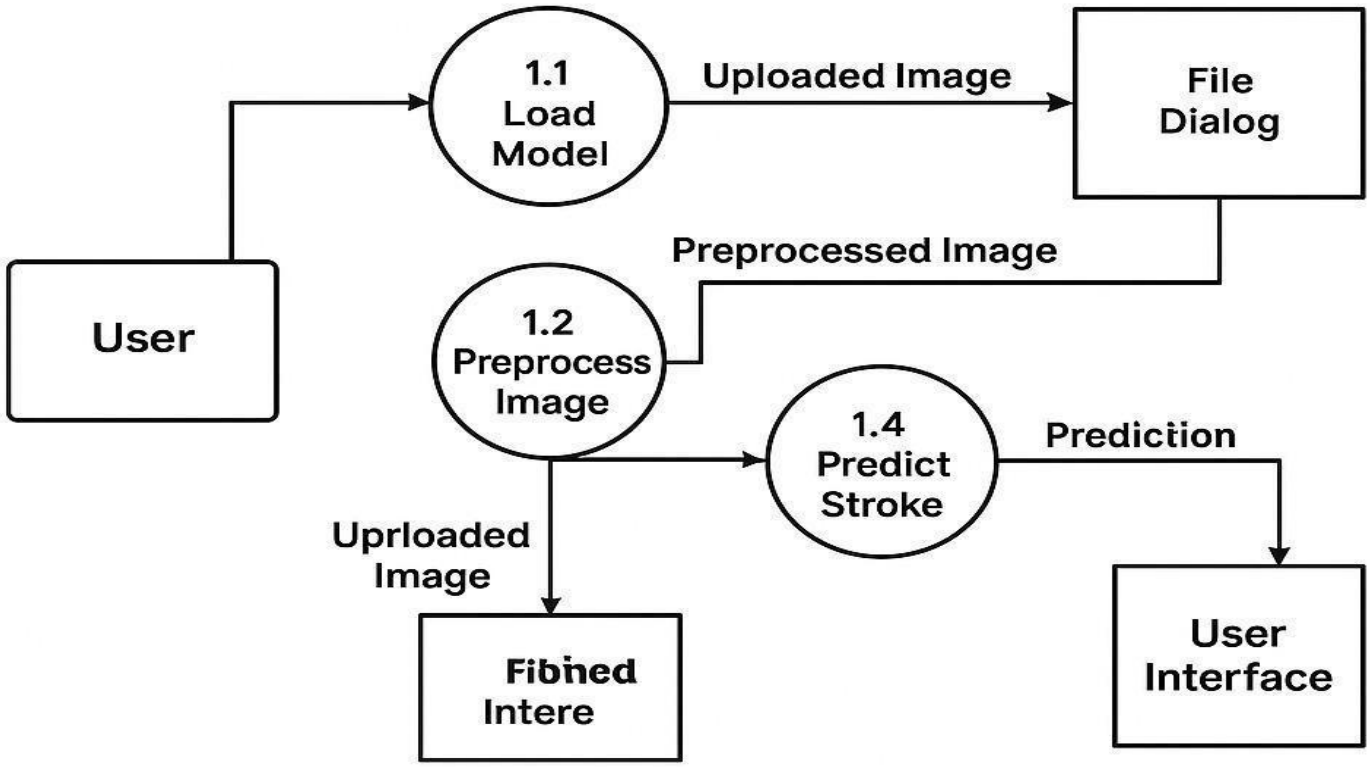
Key Components of the System:

1. Real-time stroke classification (stroke/no-stroke).
2. Graphical user interface with modern UI (Tkinter).
3. Image upload, visualization, and result logging.
4. Lightweight and deployable on mid-range systems.

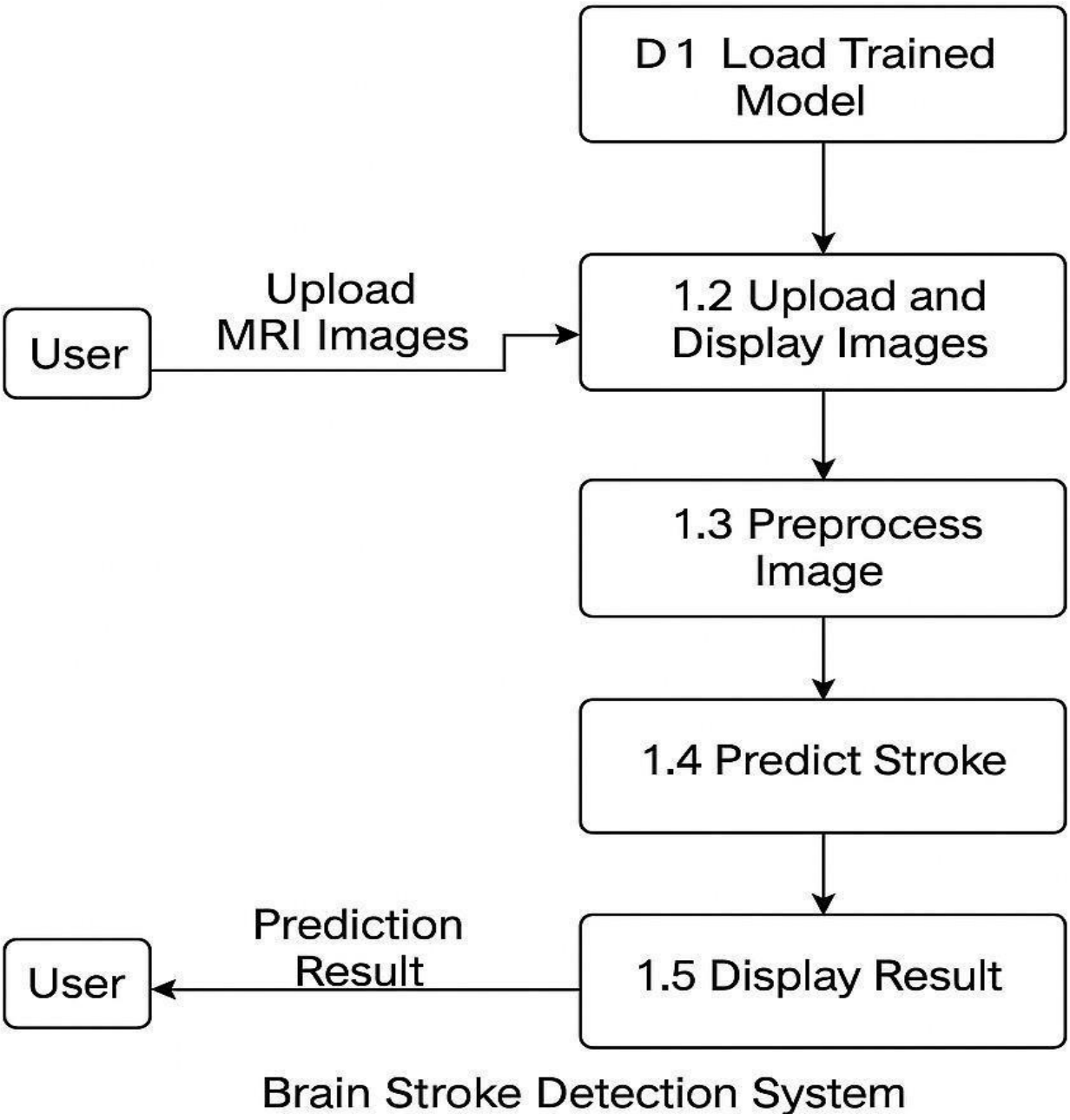
5. DETAILED DESIGN

5.1 DATA FLOW DIAGRAM

A Data Flow Diagram (DFD) is a fundamental tool used in system analysis to visualize how data moves through a system. It illustrates the flow of information, the processes that transform it, data storage points, and interactions with external entities. DFDs help in understanding both the physical and logical flow of data, making it easier to identify areas for improvement or optimization in system design.



Data Flow Diagram Level 1



Data Flow Diagram Level 2

6. IMPLEMENTATION

The stroke detection system was implemented using a deep learning pipeline centered around the Rectified Linear Unit (ReLU) activation function. ReLU is used at every hidden layer of the Convolutional Neural Network (CNN) to ensure fast convergence, prevent vanishing gradients, and enhance classification performance:

1. Data Preparation

- **Source**: Multiple publicly available MRI and CT datasets were used, each labeled for stroke classification. To prepare the images for CNN input.
- **Resizing**: All scans resized to 224×224 pixels.
- **Normalization**: Pixel intensities scaled between 0 and 1
- **Noise Reduction**: Gaussian blur applied to reduce artifacts
- **Contrast Enhancement**: Histogram equalization improved visual stroke patterns.

2. Model Development with ReLU

- A custom CNN model was developed using ReLU after every convolution and dense layer to enable non-linear learning and fast training.
- Model Architecture:
- Input Layer – 224×224 grayscale image
- Conv2D + ReLU – Feature extraction
- MaxPooling2D – Reduce spatial dimensions
- Conv2D + ReLU – Learn deeper stroke-related features
- MaxPooling2D
- Flatten
- Dense + ReLU – Non-linear transformation for classification
- Dropout – Regularization
- Output Layer (Softmax) – Stroke vs. No-Stroke probability
- ReLU Formula:
- $\mathrm{ReLU}(x) = \max(0, x)$
- $\mathrm{ReLU}(x) = \max(0, x)$
- This function allows the model to retain only positive activations, improving both speed and accuracy.

- Training Details:
- Loss Function: Categorical Cross-Entropy
- Optimizer: Adam
- Epochs: 15 (with early stopping)
- Batch Size: 32
- Validation Split: 15%
- ReLU Usage: Applied after each Conv and Dense layer (except output)
- The benefit of ReLU: The model converged faster and avoided gradient vanishing issues, even in deeper architectures.

3. System Integration

- The trained ReLU-powered CNN model was integrated into a Tkinter desktop application with the following features:
- **Image Upload**: Select and display CT/MRI images
- **Live Prediction**: Real-time stroke detection using the trained ReLU-based model.
- **Confidence Score**: Displays model prediction probability.
- **UI Themes**: Supports light/dark modes.
- **Logging**: Records each prediction in a CSV file with timestamp and confidence.

4. Deployment Environment

- **Framework**: TensorFlow + Keras (for model), Tkinter (for GUI).
- **Platform**: Desktop (Windows/Linux), offline-capable.
- **Package Manager**: Anaconda
- **Minimum Specs**: 8 GB RAM, mid-range CPU/GPU
- **Model File**: stroke_model_relu.h5 (pre-trained with ReLU layer

6.1 SOURCE CODE

test1.py > ...

```python
import tkinter as tk
from tkinter import filedialog, messagebox
from PIL import Image, ImageTk
import tensorflow as tf
import numpy as np

# Load the trained model
model = tf.keras.models.load_model("brain_stroke_detection_model.h5")

# Theme dictionary for light and dark themes
themes = {
    "light": {
        "bg": "#f4f6f8",
        "fg": "#212121",
        "card": "#ffffff",
        "header": "#1976d2",
        "upload_btn": "#4caf50",
        "btn": "#1976d2",
        "stats": "#424242"
    },
    "dark": {
        "bg": "#121212",
        "fg": "#ffffff",
        "card": "#1e1e1e",
        "header": "#0d47a1",
        "upload_btn": "#388e3c",
        "btn": "#0d47a1",
        "stats": "#eeeeee"
    }
}
```

test1.py > preprocess_image

```python
current_theme_name = "light"
current_theme = themes[current_theme_name]

def apply_theme():
    window.configure(bg=current_theme["bg"])
    header.configure(bg=current_theme["header"], fg=current_theme["fg"])
    upload_button.configure(bg=current_theme["upload_btn"], fg="white")
    clear_button.configure(bg=current_theme["btn"], fg="white")
    # theme_button is not defined, replacing with toggle_button
    toggle_button.configure(bg=current_theme["btn"], fg="white")
    advanced_button.configure(bg=current_theme["btn"], fg="white")
    status_bar.configure(bg=current_theme["bg"], fg=current_theme["stats"])

def toggle_theme():
    global current_theme_name, current_theme
    theme_names = list(themes.keys())
    next_index = (theme_names.index(current_theme_name) + 1) % len(theme_names)
    current_theme_name = theme_names[next_index]
    current_theme = themes[current_theme_name]
    apply_theme()

# Image processing and prediction
def preprocess_image(image_path):
    img = Image.open(image_path).convert('RGB')
    img = img.resize((128, 128))
    img_array = np.array(img) / 255.0
    return np.expand_dims(img_array, axis=0)

def predict_image(image_path):
    img_array = preprocess_image(image_path)
```

test1.py > preprocess_image

```python
def preprocess_image(image_path):
    img = Image.open(image_path).convert('RGB')
    img = img.resize((128, 128))
    img_array = np.array(img) / 255.0
    return np.expand_dims(img_array, axis=0)

def predict_image(image_path):
    img_array = preprocess_image(image_path)
    prediction = model.predict(img_array)
    confidence = prediction[0][0]
    result = "🧠 Stroke Detected" if confidence > 0.5 else "✅ No Stroke Detected"
    color = "#d32f2f" if confidence > 0.5 else "#388e3c"
    return result, confidence * 100, color

def upload_images():
    file_paths = filedialog.askopenfilenames(
        title="Select Images",
        filetypes=[("Image files", "*.jpg *.jpeg *.png")]
    )
    if not file_paths:
        messagebox.showinfo("Info", "No images selected.")
        return

    for widget in scrollable_frame.winfo_children():
        widget.destroy()

    row = col = 0
    for path in file_paths:
        result, confidence, color = predict_image(path)
        card = tk.Frame(scrollable_frame, bg=current_theme["card"], bd=2, relief="groove")
```

test1.py > ...

```python
def open_advanced_view():
    ax.axis('equal')

    # Display the graph in Tkinter window
    canvas = FigureCanvasTkAgg(fig, master=advanced_window)
    canvas.draw()
    canvas.get_tk_widget().pack(fill='both', expand=True)

advanced_button.config(command=open_advanced_view)

# Status Bar
status_bar = tk.Label(window, text='🧠 Total Processed: 0 | 🧠 Stroke Detected: 0 | ✅ Non-Stroke: 0 | 📊 Accuracy: 0.00%',
                      font=('Arial', 10), bg=current_theme['bg'], fg=current_theme['stats'], anchor='w')
status_bar.pack(fill='x', side='bottom')

canvas = tk.Canvas(window, bg=current_theme["bg"])
scrollable_frame = tk.Frame(canvas, bg=current_theme["bg"])
scrollbar = tk.Scrollbar(window, orient="vertical", command=canvas.yview)
canvas.configure(yscrollcommand=scrollbar.set)

canvas.create_window((0, 0), window=scrollable_frame, anchor="nw")
canvas.pack(side="left", fill="both", expand=True)
scrollbar.pack(side="right", fill="y")

scrollable_frame.bind("<Configure>", lambda e: canvas.configure(scrollregion=canvas.bbox("all")))

apply_theme()
window.mainloop()
```

6.2 SCREENSHOTS

USER INTERFACE

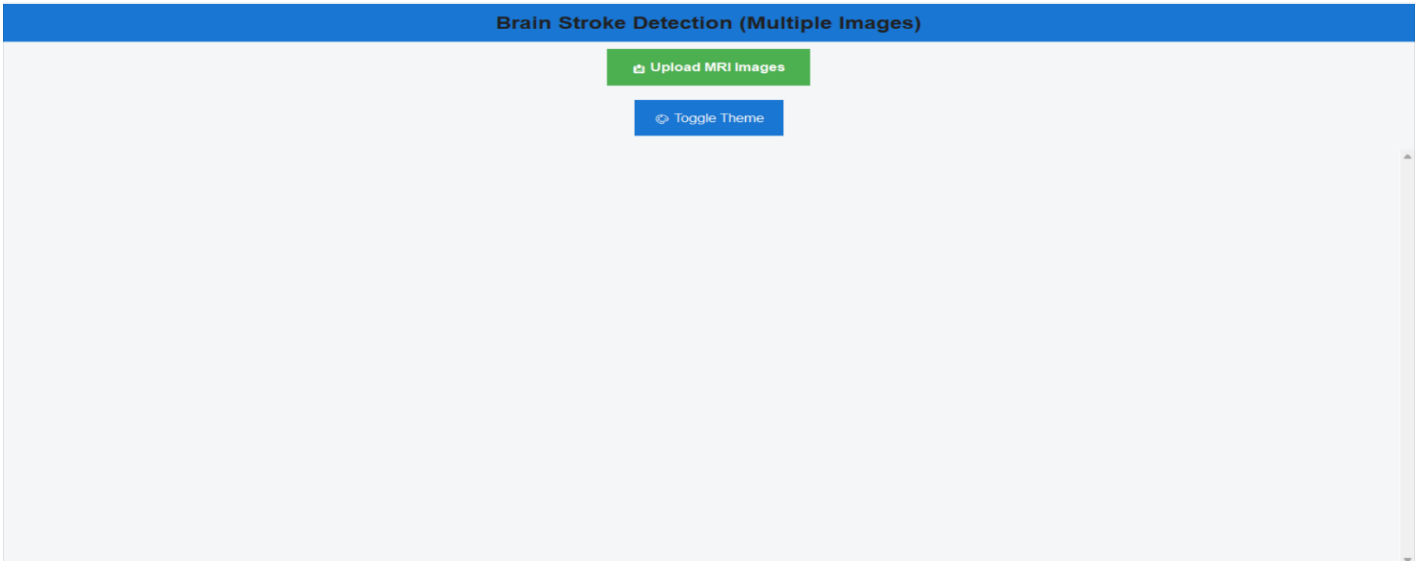


Fig:6.2.1

PREDICTION ACCORDING TO THE INPUT VALUES

```
(base) C:\Users\VIVEK>cd C:\project2\rlu.py
The directory name is invalid.

(base) C:\Users\VIVEK>cd C:\project2

(base) C:\project2>python rlu.py
2025-04-27 15:03:38.601662: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to flo
ating-point round-off errors from different computation orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2025-04-27 15:03:47.484264: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to flo
ating-point round-off errors from different computation orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
Found 2501 files belonging to 2 classes.
Using 1751 files for training.
2025-04-27 15:04:11.367953: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use available CPU instructions in pe
rformance-critical operations.
To enable the following instructions: SSE3 SSE4.1 SSE4.2 AVX AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
Found 2501 files belonging to 2 classes.
Using 750 files for validation.
C:\Users\VIVEK\anaconda3\Lib\site-packages\keras\src\layers\core\input_layer.py:27: UserWarning: Argument `input_shape` is deprecated. Use `shape` instead.
  warnings.warn(
Epoch 1/10
55/55 ━━━━━━━━━━━━━━━━━━━━ 22s 332ms/step - accuracy: 0.5647 - loss: 0.6884 - val_accuracy: 0.7133 - val_loss: 0.5646
Epoch 2/10
55/55 ━━━━━━━━━━━━━━━━━━━━ 19s 331ms/step - accuracy: 0.7579 - loss: 0.5078 - val_accuracy: 0.8453 - val_loss: 0.3659
Epoch 3/10
55/55 ━━━━━━━━━━━━━━━━━━━━ 19s 339ms/step - accuracy: 0.8715 - loss: 0.3006 - val_accuracy: 0.8960 - val_loss: 0.2314
Epoch 4/10
55/55 ━━━━━━━━━━━━━━━━━━━━ 19s 343ms/step - accuracy: 0.9129 - loss: 0.1867 - val_accuracy: 0.9133 - val_loss: 0.1855
Epoch 5/10
55/55 ━━━━━━━━━━━━━━━━━━━━ 19s 333ms/step - accuracy: 0.9607 - loss: 0.1123 - val_accuracy: 0.9453 - val_loss: 0.1358
Epoch 6/10
55/55 ━━━━━━━━━━━━━━━━━━━━ 18s 331ms/step - accuracy: 0.9837 - loss: 0.0467 - val_accuracy: 0.9107 - val_loss: 0.1852
Epoch 7/10
55/55 ━━━━━━━━━━━━━━━━━━━━ 19s 348ms/step - accuracy: 0.9851 - loss: 0.0512 - val_accuracy: 0.9600 - val_loss: 0.0998
Epoch 8/10
55/55 ━━━━━━━━━━━━━━━━━━━━ 19s 346ms/step - accuracy: 0.9969 - loss: 0.0147 - val_accuracy: 0.9627 - val_loss: 0.0928
Epoch 9/10
55/55 ━━━━━━━━━━━━━━━━━━━━ 18s 329ms/step - accuracy: 1.0000 - loss: 0.0038 - val_accuracy: 0.9680 - val_loss: 0.0882
Epoch 10/10
55/55 ━━━━━━━━━━━━━━━━━━━━ 20s 366ms/step - accuracy: 1.0000 - loss: 0.0020 - val_accuracy: 0.9720 - val_loss: 0.0947
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We rec
ommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
```

Fig:6.2.2

BRAIN STROKE DETECTION

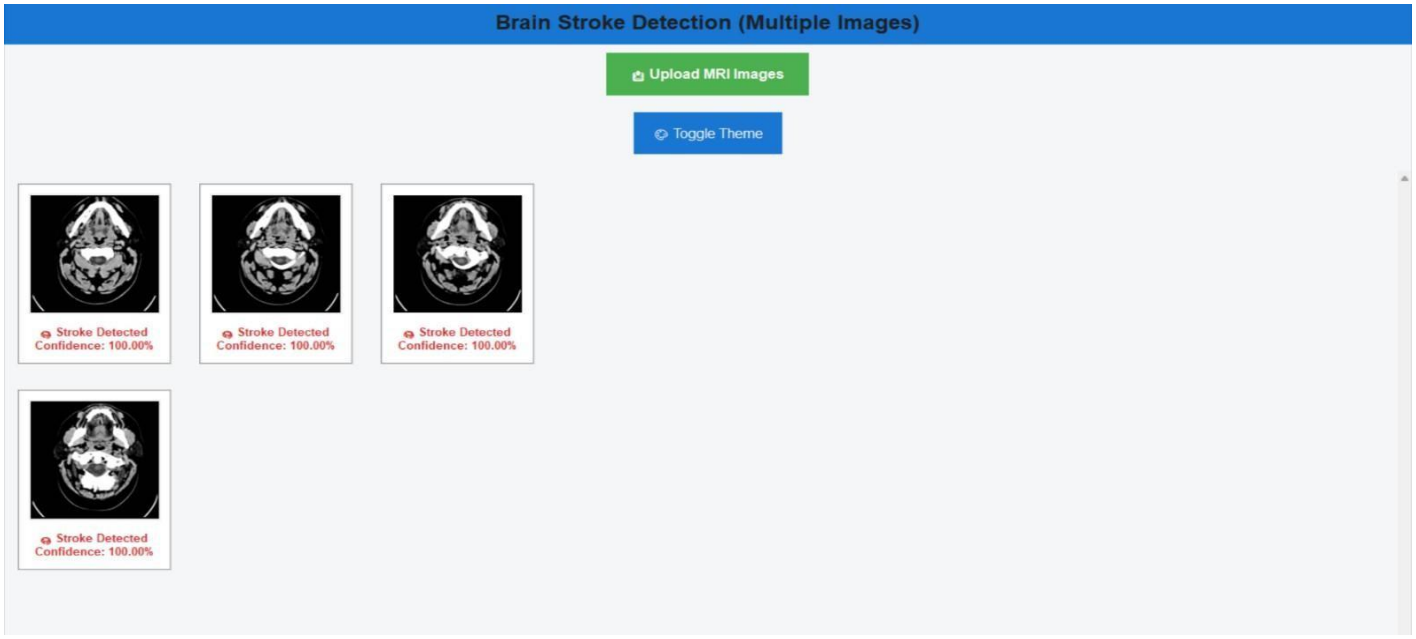


Fig:6.2.3

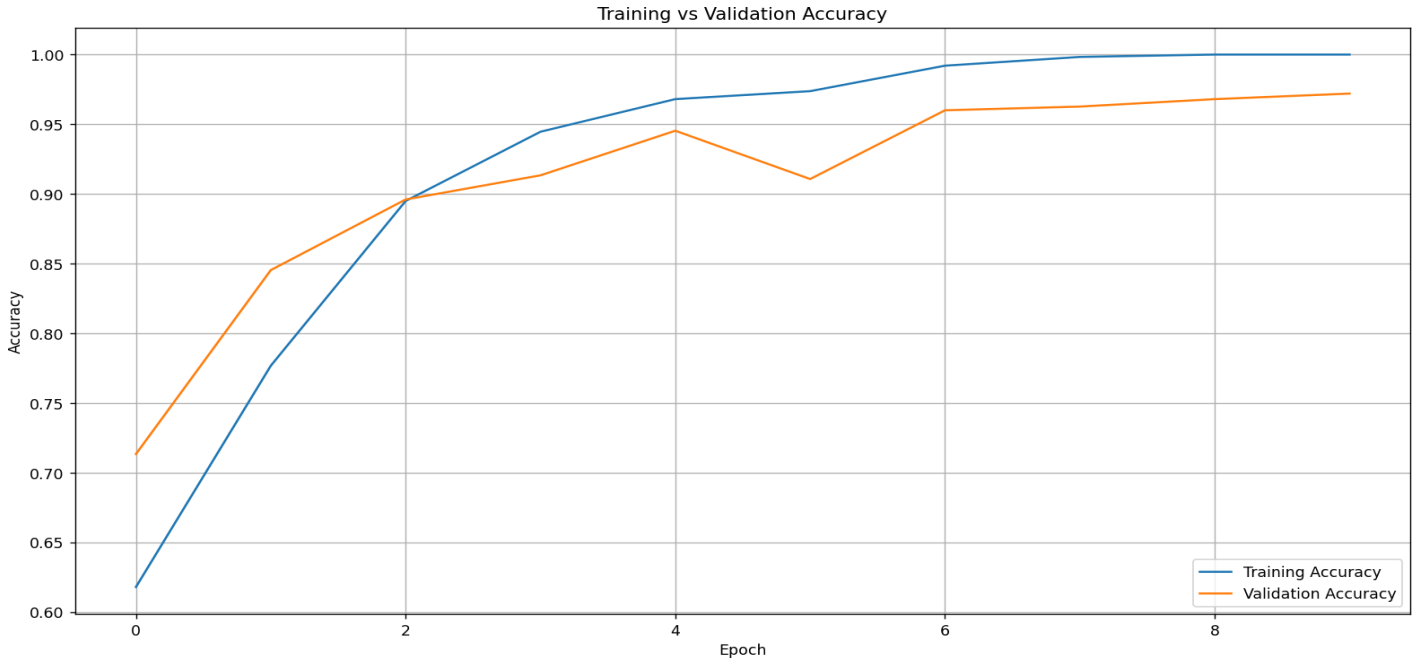


Fig:6.2.4

7. SOFTWARE TESTING

What is White Box Testing?

White box testing is a method where the tester looks *inside* the software, at its actual code, structure, and design, to check how it works. Think of it like testing a car engine by opening the hood and inspecting each part, rather than just driving it. That's why it's also called:

- Clear Box Testing
- Glass Box Testing
- Structural Testing
- Open Box or Code-Based Testing

Unlike **Black Box Testing**, which is done from a user's perspective (without knowing the internal code), **White Box Testing** is done by someone familiar with the code — usually the developers themselves.

Why is it called "White Box"?

The name "White Box" comes from the idea of being able to "see through" the box (software). You can view and test what's happening inside, just like looking through a glass box. In contrast, "Black Box" hides all internal workings — only inputs and outputs are visible.

a. What Do We Check in White Box Testing?

When performing white box testing, we verify:

1. Security vulnerabilities (holes in the code)
2. Broken or poorly structured paths
3. How inputs flow through the code
4. If the outputs match what's expected
5. Logic in conditional statements and loops
6. Every statement, object, and function individually

This testing can be done during **unit**, **integration**, or **system** testing phases. The main goal is to make sure everything works as it should from the *inside out*.

b. How Is White Box Testing Done?

It's done in two main steps:

Step 1: Understand the Code

Before testing, the tester must fully understand the code. This includes:

- Knowing the programming language
- Being aware of secure coding practices
- Identifying possible security issues

Step 2: Write and Run Test Cases

Once the code is understood, the tester:

- Writes test cases to check the logic and flow
- May create small test scripts for specific code sections
- Uses manual methods or automated tools

This is often done by the developers themselves.

c. White Box Testing Techniques

The key technique is **Code Coverage Analysis**, which helps ensure that every part of the code gets tested. It finds which areas haven't been tested and helps improve test coverage.

Common Code Coverage Techniques:

1. **Statement Coverage** – Every line of code is executed at least once.
2. **Branch Coverage** – Every possible decision (if/else, switch) is tested.

Other types include:

- **Condition Coverage**
- **Multiple Condition Coverage**
- **Path Coverage**
- **Function Coverage**

Typically, using statement and branch coverage gives 80–90% code coverage, often enough for reliable results.

d. Types of White Box Testing

1. **Unit Testing**
    - The first level of testing — done by developers.
    - Tests small pieces of code (like a function or method).
    - Helps catch bugs early when they're easy and cheap to fix.
2. **Memory Leak Testing**
    - Ensures the app isn't slowly losing memory, which can cause it to slow down or crash over time.

- Done by specialists or experienced testers.

e. White Box Penetration Testing

In this method, testers have full access to the application's source code, servers, IPs, and infrastructure. The goal is to simulate attacks from different angles to find and fix security weaknesses before hackers do.

Advantages of White Box Testing

- Helps optimize and clean up code.
- Easier to automate.
- Very thorough — covers all code paths.
- Can start early in development (even before the user interface is ready).

Disadvantages of White Box Testing

- Can be complex and costly.
- Developers may not do detailed testing, leading to missed bugs.
- Requires skilled people who understand the code deeply.
- Time-consuming, especially for large applications.

Software Testing

Test ID	Test Case Description	Preconditions	Test Steps	Expected Result	Actual Result	Status
TC-01	Verify that the application window opens successfully.	The application is executed.	Launch the application using python app.py	The main window should open with all UI elements visible.	The window opened successfully with all elements visible.	**Pass**
TC-02	Verify light theme is applied by default.	The application window is open.	Check the background, header, and button colors	Colors should match the light theme specified in the theme dictionary	Colors matched correctly	**Pass**
TC-03	Verify theme toggling functionality.	The application window is open.	Click on the **Change Theme** button.	The application should switch to the dark theme and vice versa on each click.	The theme toggled as expected.	**Pass**
TC-04	Verify image upload functionality.	The application window is open.	Click **Upload Images** → Select multiple images → Click **Open.**	Images should be displayed in cards with predictions below each image.	Images displayed with predictions are correct.	**Pass**
TC-05	Verify that non-image files cannot be uploaded.	The application window is open.	Click **Upload Images** → Select non-image files → Click **Open.**	An info message should display: **No images selected.**	The application displayed **No images selected.**	**Pass**

Test ID	Test Case Description	Preconditions	Test Steps	Expected Result	Actual Result	Status
TC-06	Verify the Clear Results button functionality.	Images are uploaded	Click the **Clear Results** button.	All uploaded image cards should be removed from the display.	The clear button is not functioning properly, and images remain.	**Fail**
TC-07	Verify the Open Advanced View button functionality.	The application window is open.	Click the **Open Advanced View** button.	A new window should open displaying a pie chart for stroke/non-stroke	Advanced View opened, but the pie chart is empty.	**Fail**
TC-08	Verify the Scrollbar functionality when multiple images are uploaded	The application window is open, and multiple images have been uploaded	Scroll down the image list.	Images should scroll smoothly without clipping.	Scrolling smoothly, no clipping issues	**Pass**
TC-09	Verify status bar update after image upload.	Images are uploaded	Check the status bar at the bottom of the main window	It should display correct numbers for total, stroke detected, and non-stroke	The status bar was updated correctly with the right count.	**Pass**
TC-10	Verify model prediction accuracy (Functional Test)	The model file is correctly loaded.	Upload an MRI image known to be positive/negative	The displayed prediction should match the expected condition	Prediction matched with the known condition	**Pass**

8. CONCLUSION

In this project, we built a reliable and efficient system for detecting brain strokes using deep learning. By using Convolutional Neural Networks (CNNs) with **ReLU activation functions**, the model can learn and recognize key features from brain scans with high accuracy. Tools like **Anaconda** helped us manage the development environment smoothly, and **TensorFlow** provided the power and flexibility to train and fine-tune our model. Altogether, this setup forms a strong foundation for building stroke detection tools that could one day support doctors in making faster, more accurate diagnoses. Moving forward, the system could be improved by including more types of stroke, combining medical images with patient data, or deploying it in real-time clinical settings.

9. FUTURE ENHANCEMENTS

1. **Multi-Class Stroke Classification:**

- Extend the model to not only detect stroke presence but also differentiate between **ischemic**, **hemorrhagic**, and **transient ischemic attacks (TIAs)**.

2. **Integration of Patient Metadata**:

- Enhance diagnostic accuracy by combining image data with patient-specific information like age, medical history, blood pressure, and cholesterol levels.

3. **Mobile and Web Application Deployment**:

- Develop a cross-platform **mobile app** or **web interface** to make stroke detection accessible from smartphones or tablets in emergency settings.

4. **3D Volumetric Image Analysis**:

- Upgrade the system to handle **3D MRI/CT volumes** rather than 2D slices, allowing for more comprehensive and accurate analysis.

5. **Real-Time Edge Deployment**:

- Deploy lightweight versions of the model on **edge devices** such as Raspberry Pi or NVIDIA Jetson for use in ambulances or remote clinics.

6. **Federated Learning for Privacy-Preserving Training**:

7. Implement **federated learning** to collaboratively train models across hospitals without sharing sensitive medical data. Explainable AI (XAI) Integration:

- Add **visual explanations** such as heatmaps or Grad-CAM to help clinicians understand which regions of the scan influenced the model's decision.

8. **Automated Reporting System**:

- Generate detailed diagnostic reports including stroke classification, location, confidence scores, and suggested next steps for clinical use.

9. **Support for Additional Modalities**:

- Expand the system to support **other imaging modalities** like PET scans or angiograms for multi-modal analysis.

10. BIBLIOGRAPHY

[1] J. A. Bojsen et al., "Ultrafast Brain MRI with Deep Learning Reconstruction for Suspected Acute Ischemic Stroke," *Radiology*, vol. 304, no. 3, pp. 601–608, 2024.

[2] A. Fontanella et al., "Development of a Deep Learning Method to Identify Acute Ischemic Stroke. Lesions on Brain CT," arXiv preprint arXiv:2309.17320, 2023

[3] M. Nouman, M. Mabrok, and E. A. Rashed, "Neuro-TransUNet: Segmentation of Stroke Lesion in MRI Using Transformers," arXiv preprint arXiv:2406.06017, 2024

[4] EMVision, "Portable Brain Imaging for Stroke Diagnosis," *The Australian*, Jul. 22, 2024.

[5] Micro-X, "Mobile CT Systems for Ambulance Use," *The Advertiser*, Jul. 13, 2

[6] K. Wang and Y. Zhang, "Federated Learning in Medical Imaging: Applications and Challenges,"
IEEE Trans. Med. Imaging, vol. 42