Group Member Name

Member 1 Name: Hammad Farooq
Phone number:03450710970
Intern ID: ARCH-2504-0094

Member 2 Name: Shaista Shahid
Phone number:03140890161
Intern ID:ARCH-2504-0095

Project Overview:

This project focuses on leveraging cutting-edge computer vision models, YOLO 11 for object detection and SAM2 (Segment Anything Model 2) for precise image segmentation, to identify and segment brain tumors from medical images. By integrating these models with OpenCV, we aim to streamline the detection and segmentation pipeline for real-time, accurate tumor localization.

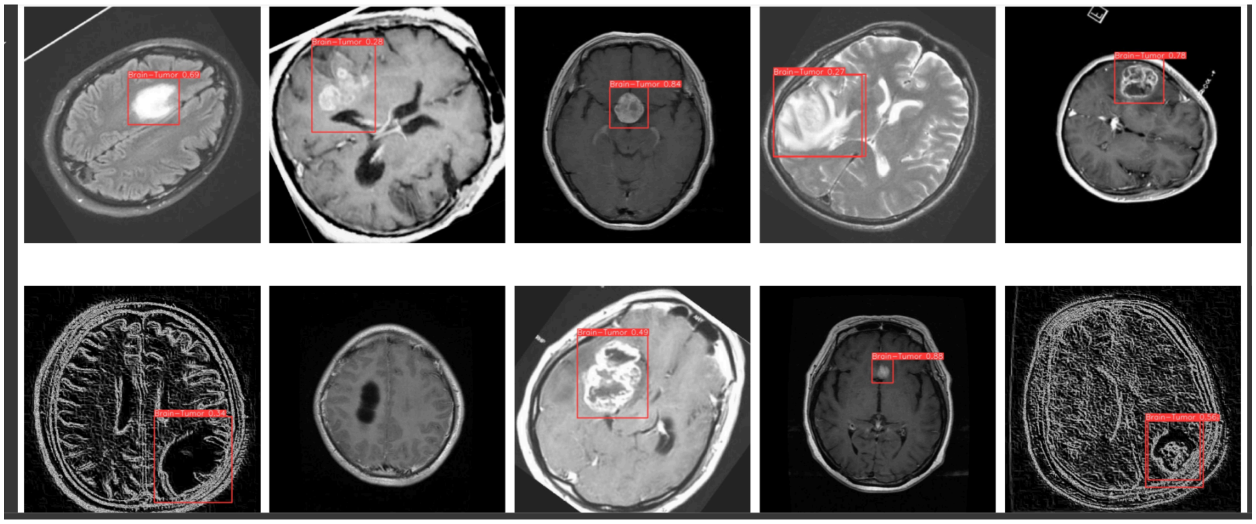
Objective:

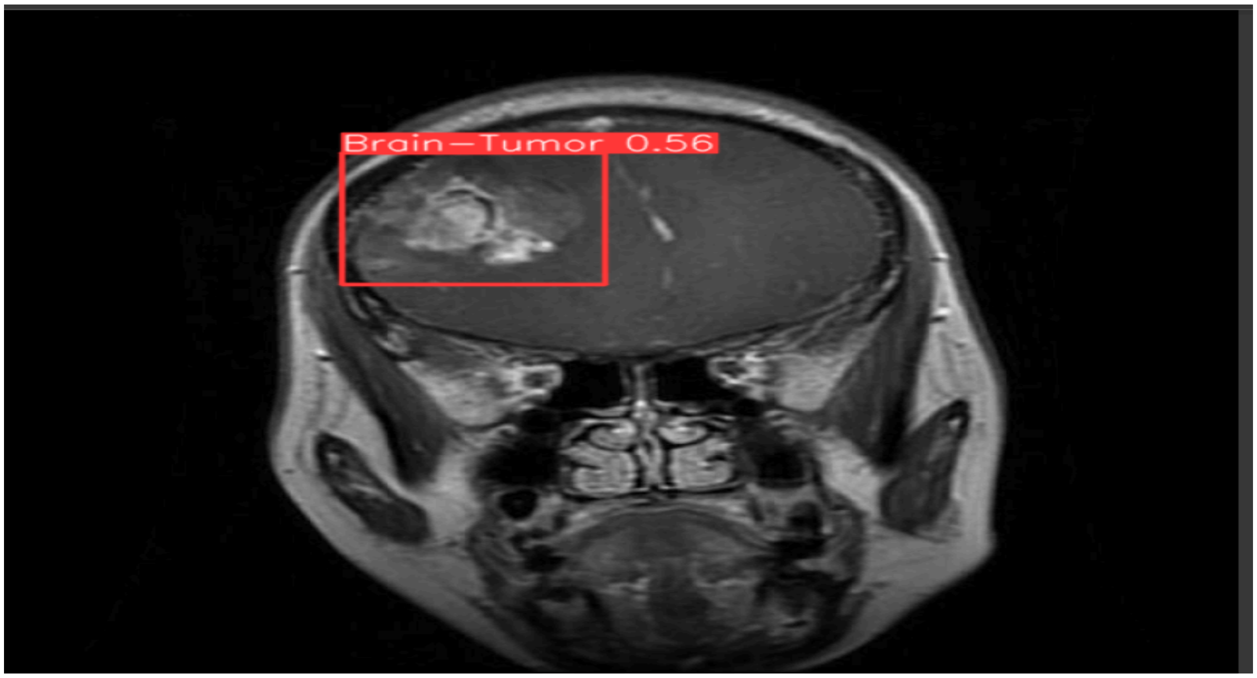
- To develop a robust system that can automatically detect and segment brain tumors in MRI images using YOLO 11 and SAM2.
- To enhance medical image analysis by providing clear visual boundaries of tumors, assisting radiologists and healthcare professionals in diagnosis.

Approach:

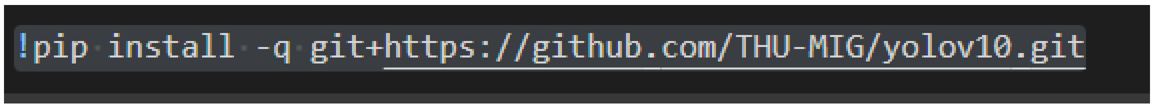
- Set up the Python environment and integrate OpenCV for image processing tasks.
- Use YOLO 11 to detect tumor regions in MRI scans.
- Apply SAM2 to accurately segment the detected tumor region for precise boundary identification.
- Train and test the models using a labeled dataset of brain tumor images.
- Evaluate performance using segmentation metrics (e.g., IoU, Dice Score) and refine the models for improved accuracy.

Diagrams, Graphs, and Visuals:



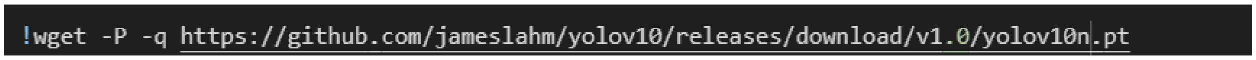


Code with Explanations :



```
!pip install -q git+https://github.com/THU-MIG/yolov10.git
```

This command installs the YOLOv10 object detection model directly from its official GitHub repository. It allows us to integrate YOLOv10 into our project for real-time brain tumor detection.

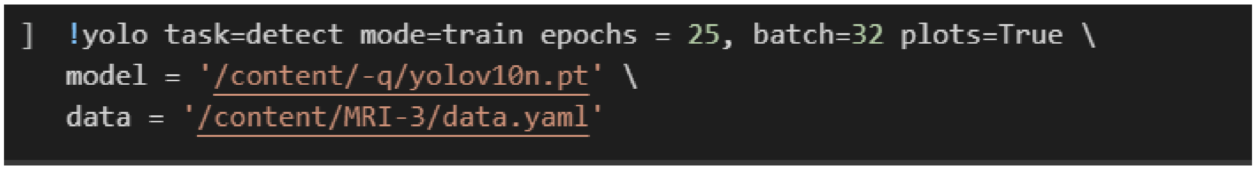


```
!wget -P -q https://github.com/jameslahm/yolov10/releases/download/v1.0/yolov10n.pt
```

This command downloads the pre-trained YOLOv10n model weights (`yolov10n.pt`) from GitHub. These weights are used to initialize the model for brain tumor detection without training from scratch.

This code connects to **Roboflow**, a platform for managing and accessing datasets for computer vision tasks.

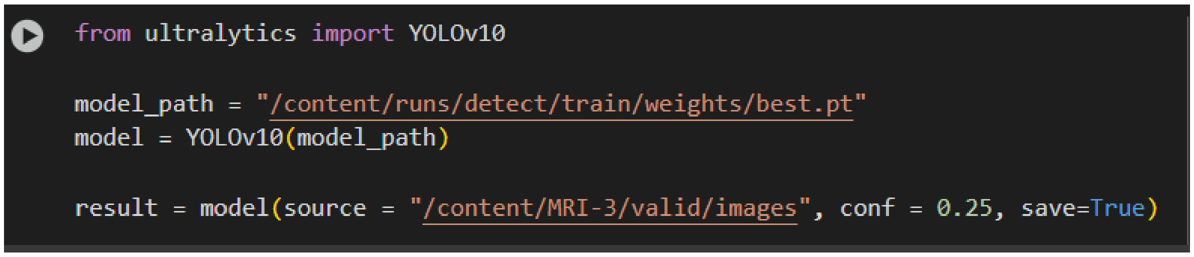
- `Roboflow(api_key = "YOUR-API-KEY")`: Authenticates your access using your API key.
- `project = rf.workspace("brain-mri").project("mri-rskcu")`: Accesses a specific brain MRI project.
- `version = project.version(3)`: Selects version 3 of the dataset.
- `dataset = version.download("yolov8")`: Downloads the dataset in YOLOv8 format, which is also compatible with YOLOv10 for tumor detection training.



```
]  !yolo task=detect mode=train epochs = 25, batch=32 plots=True \
   model = '/content/-q/yolov10n.pt' \
   data = '/content/MRI-3/data.yaml'
```

This command starts training the YOLO model for brain tumor **detection** using your dataset and a pre-trained model:

- `task=detect`: Specifies that the task is **object detection**.
- `mode=train`: Sets the mode to **training**.
- `epochs=25`: Trains the model for **25 iterations** over the dataset.
- `batch=32`: Uses a **batch size of 32** images per training step.
- `plots=True`: Enables generation of **training plots** (e.g., loss curves, accuracy).
- `model='/content/-q/yolov10n.pt'`: Loads the **YOLOv10n pre-trained weights** to fine-tune on your dataset.
- `data='/content/MRI-3/data.yaml'`: Points to the **dataset configuration file**, which includes image paths and class labels.

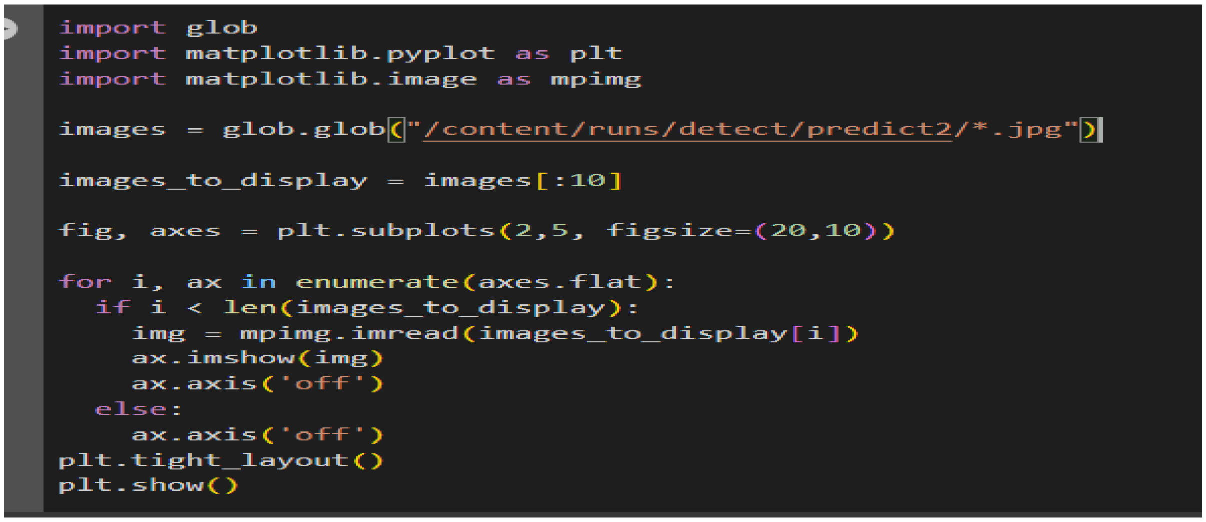


```
from ultralytics import YOLOv10

model_path = "/content/runs/detect/train/weights/best.pt"
model = YOLOv10(model_path)

result = model(source = "/content/MRI-3/valid/images", conf = 0.25, save=True)
```

This code uses the **Ultralytics YOLOv10** model to perform brain tumor detection on validation images:

- `from ultralytics import YOLOv10`: Imports the YOLOv10 model class from the Ultralytics library.
- `model_path = "/content/runs/detect/train/weights/best.pt"`: Specifies the path to the **best trained model weights** obtained during training.

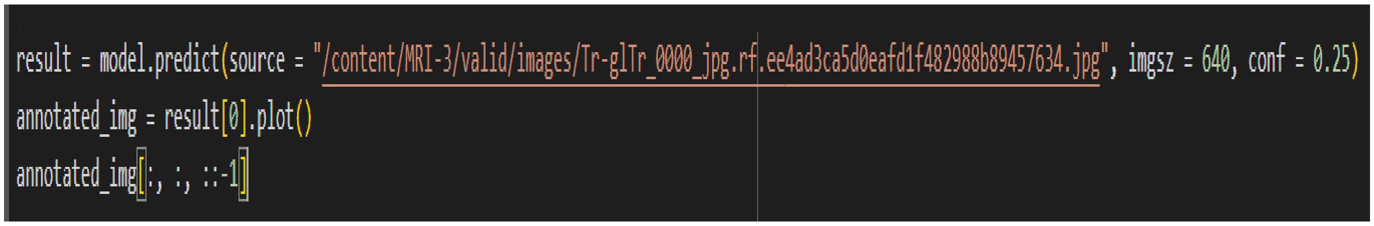
- `model = YOLOv10(model_path)`: Loads the trained YOLO10 model using those weights.
- `result = model(source="/content/MRI-3/valid/images", conf=0.25, save=True)`: Runs **inference** on the validation images:
  - `source`: Folder containing the images to test.
  - `conf=0.25`: Sets the **confidence threshold** to 25% — only predictions with confidence ≥ 0.25 will be shown.
  - `save=True`: Saves the detection results (with bounding boxes) as output images.



```
import glob
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

images = glob.glob("/content/runs/detect/predict2/*.jpg")

images_to_display = images[:10]

fig, axes = plt.subplots(2,5, figsize=(20,10))

for i, ax in enumerate(axes.flat):
  if i < len(images_to_display):
    img = mpimg.imread(images_to_display[i])
    ax.imshow(img)
    ax.axis('off')
  else:
    ax.axis('off')
plt.tight_layout()
plt.show()
```

This code is used to **display the first 10 output images** generated by the YOLOv10 model after prediction:

- `glob.glob("/content/runs/detect/predict2/*.jpg")`: Collects the paths of all `.jpg` images from the prediction output folder.
- `images_to_display = images[:10]`: Selects the **first 10 predicted images** to visualize.
- `fig, axes = plt.subplots(2,5, figsize=(20,10))`: Creates a **2x5 grid of subplots** with a large figure size for clear image display.
- The `for` loop:
  - Loads and displays each selected image using `matplotlib`.
  - Turns off axes for a clean look.
- `plt.tight_layout()` adjusts spacing between subplots.
- `plt.show()` displays the final image grid.



```
result = model.predict(source = "/content/MRI-3/valid/images/Tr-glTr_0000_jpg.rf.ee4ad3ca5d0eafd1f482988b89457634.jpg", imgsz = 640, conf = 0.25)
annotated_img = result[0].plot()
annotated_img[:, :, ::-1]
```

This code performs **tumor detection on a single image** and prepares the result for display:

- `model.predict(...)`: Uses the YOLOv10 model to **predict objects (tumors)** in the specified image.
  - `source`: Path to a **single MRI image**.
  - `imgsz=640`: Resizes the image to **640x640 pixels** before prediction (standard for YOLO).
  - `conf=0.25`: Sets the **confidence the shold** to 25%.
- `result[0].plot()`: Plots the **annotated image** with bounding boxes and labels on the first (and only) result.
- `annotated_img[:, :, ::-1]`: Converts the image from **RGB to BGR color format**, which is often needed when displaying the image using OpenCV.

Theory Tasks:

Chapter 1: The Machine Learning Landscape

Basic Terminologies related to Machine Learning

In machine learning, key terms include **features** (input variables), **labels** (target outputs), **models** (algorithms that learn patterns), **training data** (used to train the model), and **testing data** (used to evaluate performance). Other important terms include **overfitting** (when a model learns noise) and **underfitting** (when a model fails to learn patterns).

Supervised / Unsupervised / Batch / Online Learning

Supervised learning uses labeled data to train models (e.g., classification, regression), while **unsupervised learning** deals with unlabeled data to find hidden patterns (e.g., clustering). **Batch learning** trains the model using the entire dataset at once, whereas **online learning** updates the model incrementally as new data arrives.

Challenges related to Data and Algorithms

Common challenges in machine learning include **missing or imbalanced data**, **noisy inputs**, **high dimensionality**, and **bias in datasets**. Algorithm-related issues include **model complexity**, **slow training**, and difficulty in choosing the right model or **evaluation metrics** for specific tasks.

Hyperparameter Tuning and Model Selection

Hyperparameter tuning involves optimizing the settings (like learning rate, number of trees, or batch size) that control the training process but aren't learned from data. **Model selection** is the process of comparing different algorithms or configurations to find the one that performs best on validation data.

Chapter 2: End-to-End Machine Learning Project

The Machine Learning Pipeline

A machine learning pipeline is a step-by-step process that includes data collection, data preprocessing, feature engineering, model training, evaluation, and deployment. It ensures a structured workflow, making it easier to build, test, and deploy models efficiently and consistently.

Data Exploration and Visualization Techniques

Data exploration involves understanding the dataset's structure, missing values, and patterns through descriptive statistics. Visualization techniques like histograms, box plots, scatter plots, and heatmaps help identify trends, outliers, and relationships between features, guiding better decision-making for preprocessing and modeling.

Custom Transformers and Feature Scaling

Custom transformers are user-defined preprocessing tools built using libraries like `scikit-learn` to handle specific transformations (e.g., text cleaning, domain-specific encoding). Feature scaling, like StandardScaler or MinMaxScaler, is used to normalize data so that all features contribute equally to the model's learning process.

Fine-tuning Hyperparameters using GridSearchCV and RandomizedSearchCV

GridSearchCV exhaustively searches over a predefined set of hyperparameters, while RandomizedSearchCV samples a fixed number of parameter combinations randomly. Both techniques help find the best hyperparameters to improve model performance by testing different settings through cross-validation.

Chapter 2 code with explanation:

```
import numpy as np def split_train_test(data, test_ratio):
shuffled_indices = np.random.permutation(len(data))
 test_set_size = int(len(data) * test_ratio)
test_indices = shuffled_indices[:test_set_size]
 train_indices = shuffled_indices[test_set_size:]
return data.iloc[train_indices], data.iloc[test_indices
```

- Takes a dataset and a test ratio as input.
- Shuffles the dataset indices randomly.
- Calculates how many samples should be in the test set.

- Uses the first part of the shuffled indices for the test set.
- Uses the remaining indices for the training set.
- Returns both sets using `.iloc` for row selection.

```
from zlib import crc32 def test_set_check(identifier, test_ratio): return
crc32(np.int64(identifier)) & 0xffffffff < test_ratio * 2**32 def
split_train_test_by_id(data, test_ratio, id_column): ids = data[id_column]
in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio)) return
data.loc[~in_test_set], data.loc[in_test_set]
housing_with_id = housing.reset_index() # adds an `index` column train_set,
test_set = split_train_test_by_id(housing_with_id, 0.2, "index")
housing_with_id["id"] = housing["longitude"] * 1000 + housing["latitude"] train_set,
test_set = split_train_test_by_id(housing_with_id, 0.2, "id")
```

This code demonstrates a method to split a dataset into training and test sets **deterministically** using unique identifiers instead of random shuffling. It ensures that the same rows consistently go into the training or test set, even if the dataset is updated. The `test_set_check` function uses the CRC32 hash function to compute a hash of a row's ID, then checks whether that hash falls within the range defined by the test ratio. The `split_train_test_by_id` function applies this check to all rows based on an `id_column`, returning two separate datasets. Initially, it uses the DataFrame's index as a unique ID, then creates a more stable custom ID using the formula `longitude * 1000 + latitude`, ensuring consistent splits across runs and avoiding data leakage. This method is especially useful in production environments where reproducibility is critical.

Video Demonstration :

https://youtu.be/l4dgV2sbdIk

GitHub Repository:

https://github.com/CodeWithHammadFarooq/Brain_Tumor_Detection