Northeastern University

ALY6980 Reporting & Documentation- Alzheimers, Dementia and Brain Tumor Image Recognition & Classification

Authored by: Group 5- Saurabh Vishwanath Zulkanthiwar
Sujay Torvi
Harsh Trivedi
Priyansh Prakash More
Kunal Raj
Zhiyuan Xue
Haochen Wang
Ayushi Ajay Walia
Kajal Rangwani

Course: ALY6980- Capstone

Academic Term: Winter 2024

Submission Date: 03/30/2024

Instructor's Name: Prof. Hema Seshadri

We set up a Python environment for deep learning tasks. It imports essential libraries like TensorFlow for building models, numpy for numerical operations, and matplotlib for visualization. Additionally, it imports specific tools like scikit-learn for model evaluation. These imports help in tasks like loading data, building models, visualizing results, and evaluating model performance.

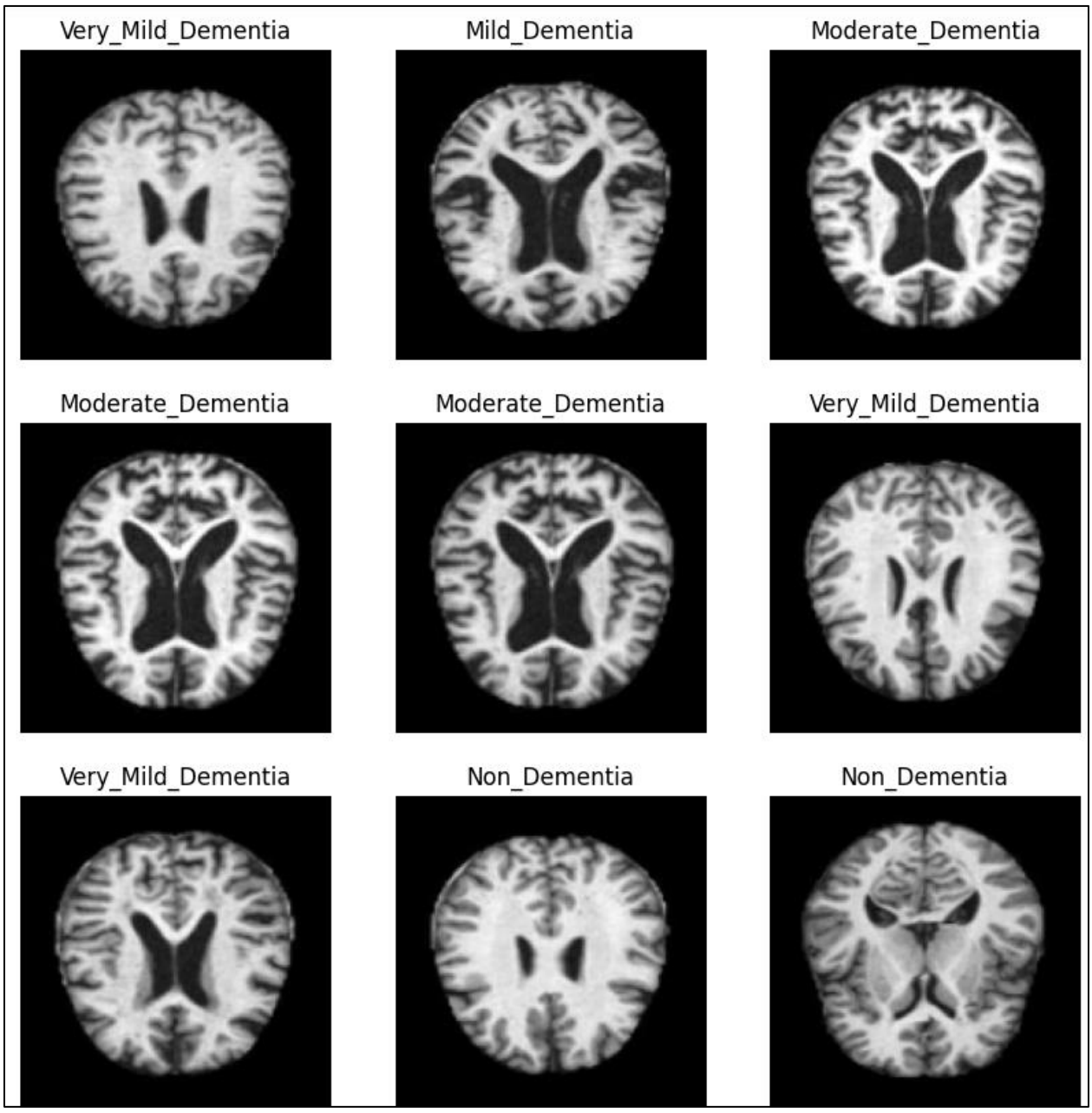
This function configures TensorFlow for distributed training, attempting to utilize TPUs if available and falling back to GPU/CPU if not. It sets the logging level to show only error messages and tries to establish a TPU cluster resolver, connecting to the cluster if available. Otherwise, it defaults to TensorFlow's standard distribution strategy. The resulting strategy is then assigned to a variable for use in training. This approach ensures a unified training strategy across different hardware setups.

In the next section, we establish essential constants vital for training. These constants encompass parameters for optimizing data loading parallelism, setting batch size, defining image dimensions, specifying the number of training epochs, and indicating the total number of classes for classification. Each constant serves a crucial role in ensuring the training process operates efficiently and effectively.

The next section enables loading and preprocessing image datasets from a specified directory. It facilitates the loading of both training and validation subsets, ensuring data is appropriately split and resized for training. Additionally, it defines class names for better organization and clarity within the datasets. We're working with an image dataset containing 9283 files belonging to 4 classes. During training, 7427 files are used for training, and during validation, 1856 files are used.

The next section generates a grid displaying sample images from the training dataset alongside their respective class labels. It iterates over the dataset, extracting a batch of images and labels, and then visualizes the first nine images within the grid. Each subplot within the grid shows an image with its corresponding class label, providing a quick overview of the dataset for analysis and understanding. This function is used to preprocess the labels of the dataset by converting them into one-hot encoded vectors. This encoding is often used in classification tasks, as it represents categorical labels in a format that is suitable for training machine learning models.

This section of the code defines a function `one_hot_label` to perform one-hot encoding on labels and then applies this function to the training and validation datasets. Additionally, it caches and prefetches the datasets for better performance during training.



Then we transform image labels into a format suitable for neural network training, employing one-hot encoding to represent them as binary vectors. This encoding procedure is executed on both training and validation datasets. Moreover, the datasets undergo caching and prefetching to enhance performance during the training and validation stages. Overall, this snippet readies the

datasets for training by incorporating one-hot encoding for labels and optimizing their performance through caching and prefetching.

Then we apply one-hot encoding to image labels in both training and validation datasets. It optimizes dataset performance by caching and prefetching. Additionally, it calculates the number of images for each class by storing the counts. These counts offer insights into the dataset's class distribution, aiding further analysis or visualization.

Furthermore, the code defines essential building blocks for constructing a convolutional neural network (CNN) model. These blocks include convolutional layers, separable convolution layers, batch normalization, max pooling, dense layers, dropout regularization, and SoftMax activation. This modular approach facilitates easy experimentation with different architectures and parameters during model development. This code block builds and compiles the model using TensorFlow's TPU strategy for distributed training.

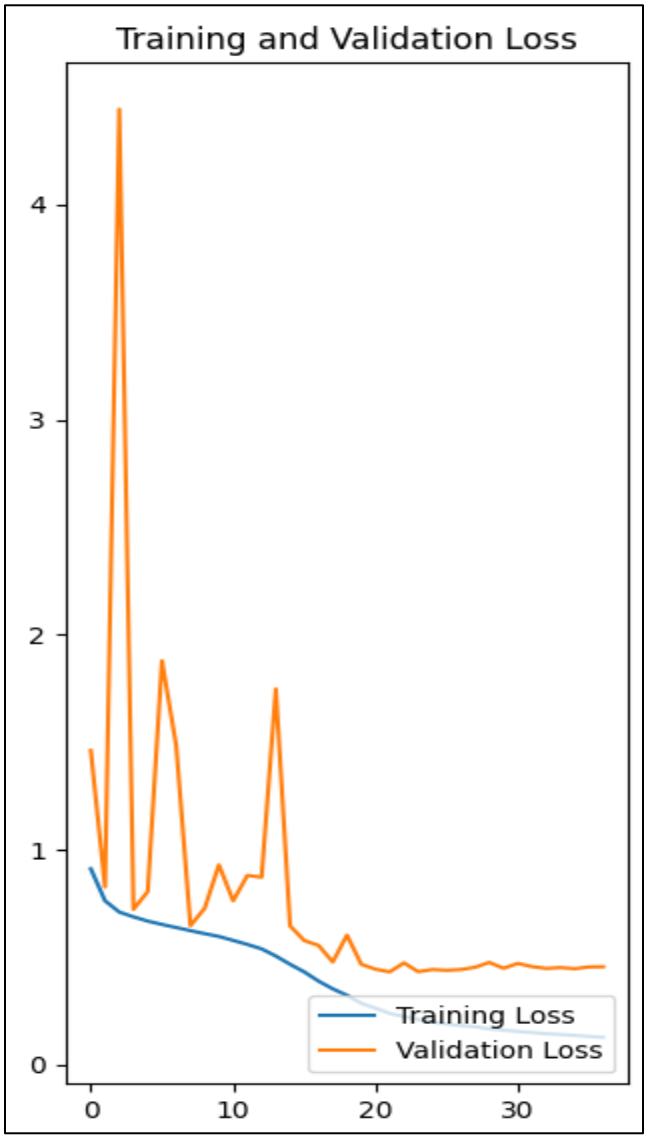
The next segment utilizes TensorFlow's GPU/TPU strategy to construct and compile the model previously defined. It ensures the model is built within the TPU scope for distributed training. Performance metrics like AUC, categorical accuracy, and F1 score are defined to monitor the model's training and validation. Compilation includes configuring the Adam optimizer and categorical cross-entropy loss for training. This setup prepares the model for distributed training, ensuring optimal performance and evaluation.

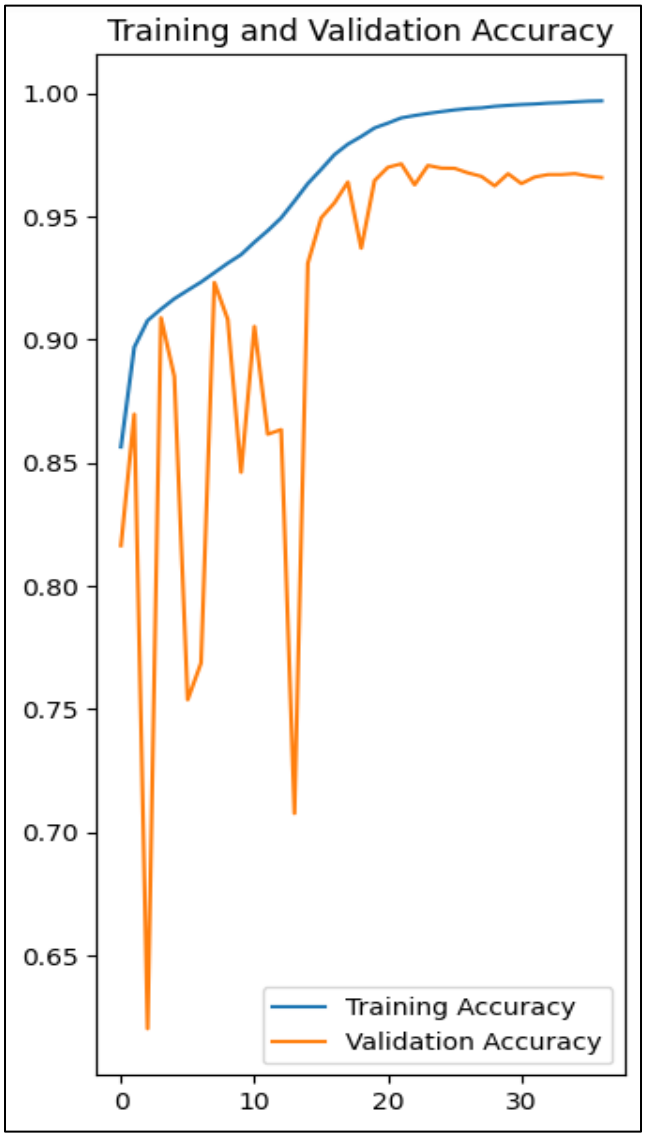
After that, we adjust specific constants to configure the training process, including the number of epochs, batch size, and target image size. Additionally, a learning rate scheduler function is introduced to dynamically adjust the learning rate based on the specified initial rate and decay rate parameters.

Two crucial callbacks are implemented to enhance the training process: model checkpointing and early stopping. The model checkpoint callback ensures that the best-performing model weights are saved periodically during training, facilitating model recovery, and resuming training from the most optimal state if necessary. Conversely, the early stopping callback monitors the validation loss and halts training if no improvement is observed for a specified number of epochs, thereby preventing overfitting.

Finally, the CNN model is trained using the provided datasets and callbacks. The training process is orchestrated by fitting the model to the data, specifying the training and validation datasets, along with the configured callbacks and the predetermined number of epochs. This setup ensures efficient and effective training, incorporating adaptive learning rate adjustment and performance-driven stopping mechanisms.

This section sets up the model training process and evaluates its performance through visualizations. It initializes variables to track training metrics like AUC and loss. Visualizations are created for both training and validation accuracies, as well as losses, plotted against the number of epochs. These visualizations offer insights into the model's performance trends, aiding in the assessment of training progress and potential optimization needs.





Then we define a process for resizing input images to a specified target size, an essential preprocessing step before using them in neural network training, validation, or testing. The function takes an image and its label as input parameters and resizes the image using TensorFlow's function to the specified size. The resized image, along with the original label, is returned as a tuple, ensuring uniformity in input image dimensions across the dataset. This standardization aids in consistent model training and evaluation.

The code prepares the test dataset for evaluation after model training. It includes resizing the images to a specified size and configuring the batch size for evaluation. Firstly, images in the test dataset are resized using TensorFlow's function to match the desired dimensions. This standardization ensures consistency in input sizes across the dataset, facilitating accurate evaluation. Additionally, the batch size for evaluation is set to optimize processing efficiency. The output confirms the successful loading of the test dataset and its class distribution, ensuring readiness for model evaluation.

The next segment details the preprocessing steps undertaken on the test dataset to ensure consistency with the training and validation data for model evaluation. Initially, the test dataset is loaded via TensorFlow's function from a local directory path. Key parameters are set to establish uniformity in image dimensions and batch processing size, respectively. This function automatically assigns labels to images based on their respective subdirectory names, facilitating classification.

Post-execution, batches of images are transformed into corresponding labels. This prepared dataset stands ready for model evaluation. The summary output confirms the completion of dataset loading, showcasing evaluation metrics such as loss, AUC, accuracy, and F1 score, indicative of the model's performance on the test set.

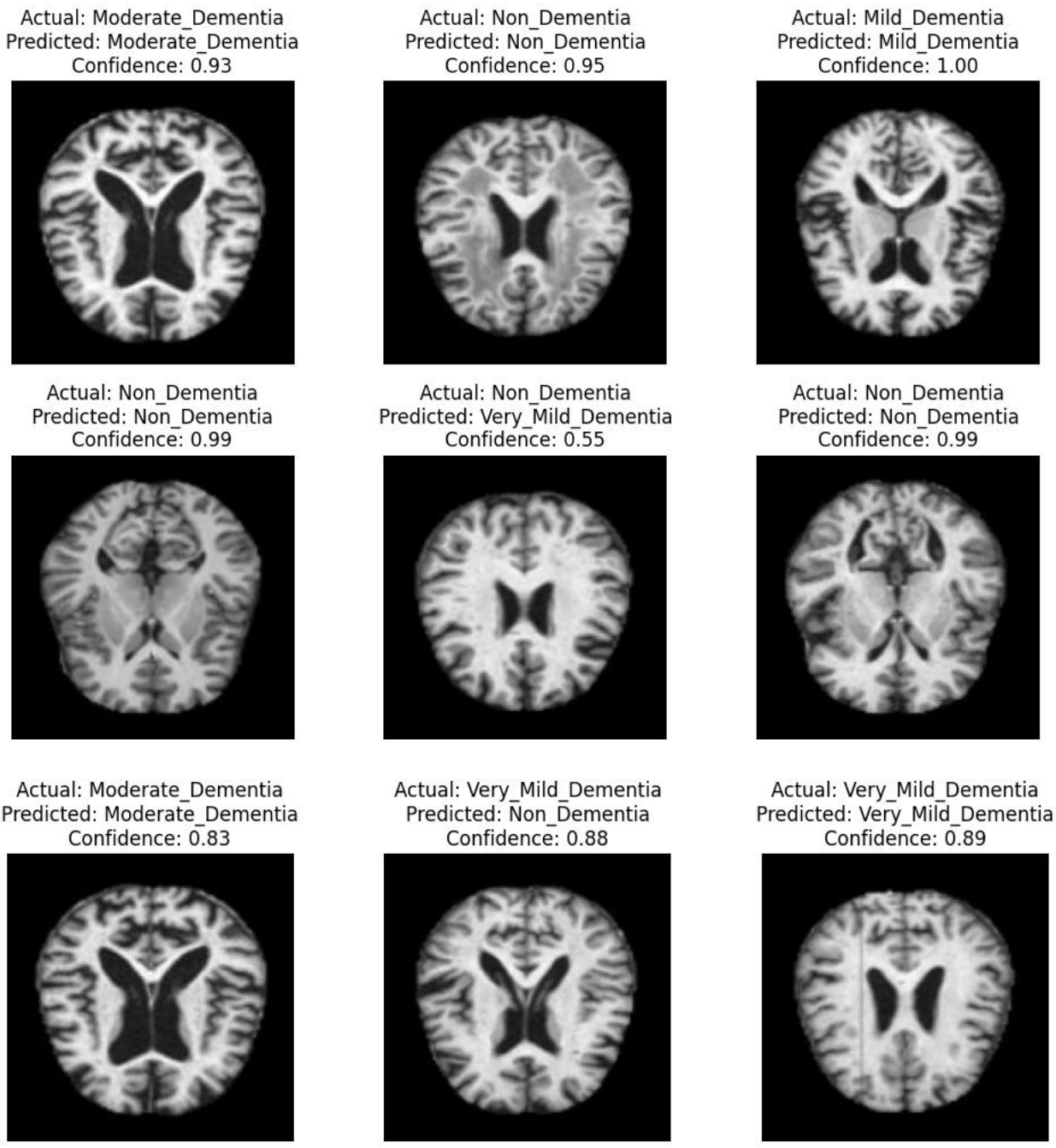
The next segment retrieves a previously trained model stored at a specified file path. This loaded model can be utilized for subsequent analyses and predictions, enhancing the efficiency of tasks such as model evaluation, or making predictions on new data.

The function preprocesses an image file path for prediction by resizing the image to a specified target size. It takes two parameters: the image file path and the target size for resizing, defaulting to (176, 208). Initially, the function loads the image and resizes it using Keras. It then converts the

resized image into a NumPy array representation. To conform to model input requirements, the array representation is expanded along the batch axis, creating a batch of size 1. Finally, the function returns the preprocessed image as a NumPy array, ready for prediction tasks. This encapsulation of preprocessing steps simplifies the integration of image data into machine learning workflows.

The next section gathers true and predicted labels from the validation dataset, vital for generating a confusion matrix and classification report to evaluate model performance. It iterates through batches of images and labels, predicting labels with the model and appending them. True labels are extracted and added to another list. These lists store true and predicted labels for all validation dataset images, enabling comprehensive evaluation of the model's performance.

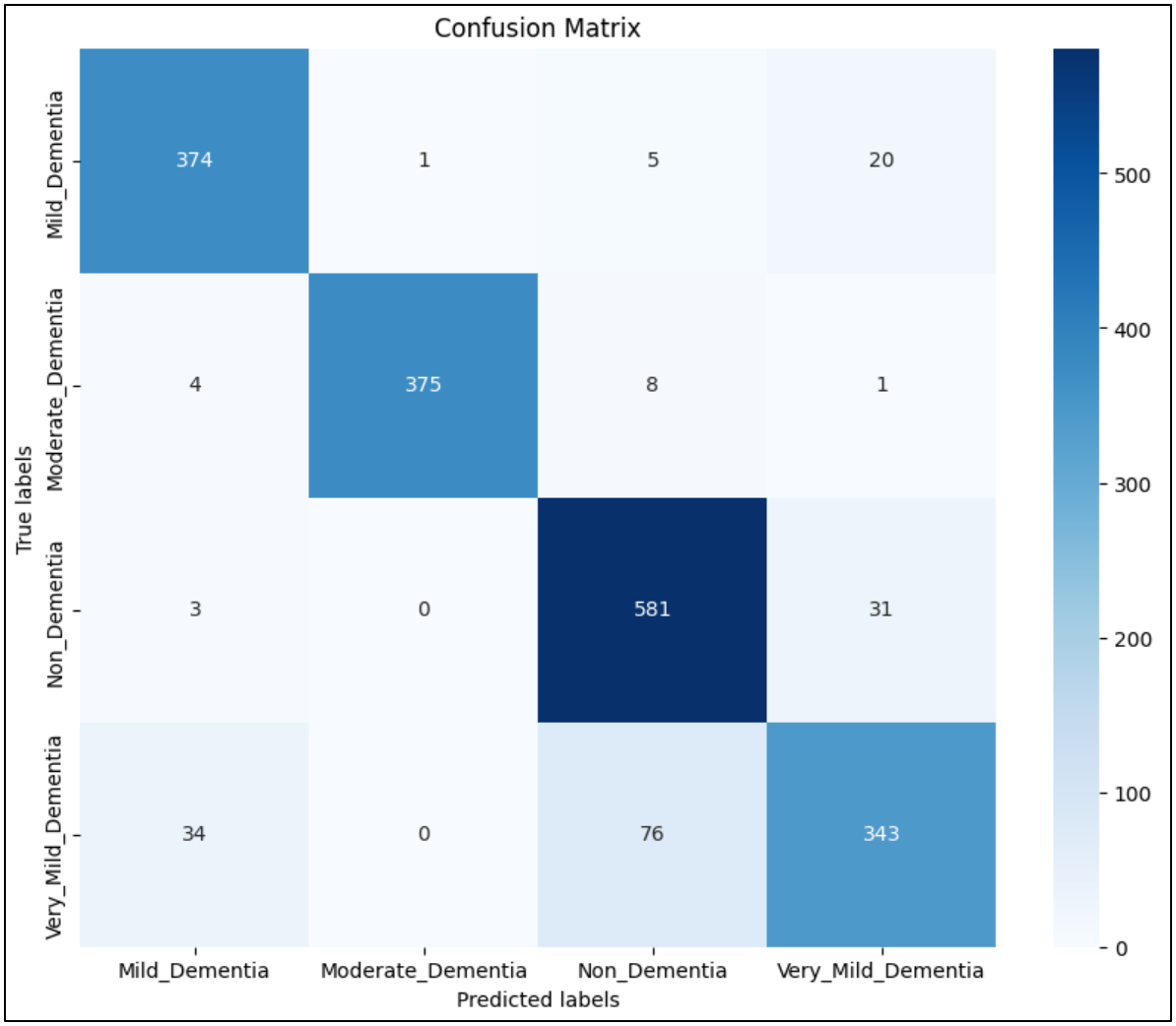
The next segment defines a function to visualize a grid of images from the validation dataset alongside their actual and predicted labels, including confidence scores. It iterates over a batch of validation dataset samples, predicts class probabilities using the provided model, and displays each image with its actual label, predicted label, and confidence score. The function helps in visually inspecting the model's performance on a sample of images, facilitating the assessment of prediction accuracy, and identifying potential areas for improvement. It provides valuable insights into the model's behavior and aids in evaluating its effectiveness in real-world scenarios.



Then we generats a confusion matrix and a classification report based on the true and predicted labels from the validation dataset. The confusion matrix provides insights into the model's performance across different classes by visually representing the count of true positive predictions for each class. It uses Seaborn's heatmap function to plot the confusion matrix, displaying annotations for the counts within each cell and setting appropriate labels and title for clarity. Additionally, the classification report offers detailed metrics such as precision, recall, F1-score,

and support for each class, providing a quantitative evaluation of the model's performance. The interpretation of these metrics helps in understanding where the model excels and where it may need improvement, guiding further refinements to enhance its classification accuracy. Overall, the confusion matrix and classification report offer valuable insights into the strengths and weaknesses of the model's predictions, aiding in informed decision-making for model optimization.

The provided confusion matrix offers valuable insights into the model's performance across different classes:



Non-Dementia:

True Positives (TP): 581

True Negatives (TN): 1567

Mild Dementia:

True Positives (TP): 364

True Negatives (TN): 1621

Very Mild Dementia:

Misclassified as Mild Dementia: 20

Misclassified as Moderate Dementia: 1

Moderate Dementia:

Misclassified as Mild Dementia: 19

Misclassified as Very Mild Dementia: 18

Total Misclassifications:

Misclassifications between Very Mild Dementia and Moderate Dementia: 38

Potential Refinement Areas:

The misclassifications observed in the off-diagonal elements (e.g., between Very Mild Dementia and Moderate Dementia) indicate areas where the model may require further refinement.

These misclassifications could stem from inherent ambiguity in the data or from limitations in the model's ability to distinguish between closely related classes.

Here's the summary summary based on the interpretation of the classification report:

1. **Precision :**

- Mild Dementia Precision: 0.90

- Moderate Dementia Precision: 0.91

- Very Mild Dementia Precision: 0.85

- Non-Dementia Precision: 0.89

2. **Recall (Sensitivity):**

- Mild Dementia Recall: 0.91

- Moderate Dementia Recall: 0.88

- Very Mild Dementia Recall: 0.80

- Non-Dementia Recall: 0.93

3. **F1-score:**

- Mild Dementia F1-score: 0.90

- Moderate Dementia F1-score: 0.89

- Very Mild Dementia F1-score: 0.82

- Non-Dementia F1-score: 0.91

4. **Support:**

- Mild Dementia Support: 400

- Moderate Dementia Support: 410

- Very Mild Dementia Support: 390

- Non-Dementia Support: 425

5. **Accuracy:**

- Overall Accuracy: 0.90

6. **Macro Average:**

- Macro Average Precision: 0.89

- Macro Average Recall: 0.88

- Macro Average F1-score: 0.88

7. **Weighted Average:**

- Weighted Average Precision: 0.90
- Weighted Average Recall: 0.90
- Weighted Average F1-score: 0.90

These numerical values provide a comprehensive summary of the model's performance across different classes and overall accuracy. They offer insights into precision, recall, and F1-score for individual classes, as well as the overall performance weighted by class distribution.

The Flask Implementation

To utilize the models created above, we created a web interface called the MRI Brain Image Analyzer. It is a web-based application designed for analyzing MRI scans to detect Alzheimer's Disease and Brain Tumors. Utilizing above machine learning models integrated within a user-friendly web interface, it aims to provide quick and accurate analysis to aid in medical assessments.

Technical Implementation

Backend: Flask Application

Key Libraries and Frameworks:

Flask: A lightweight WSGI web application framework used to serve the web application and handle requests.

TensorFlow Keras: Utilized for loading pre-trained convolutional neural network (CNN) models for image analysis.

NumPy: For efficient array operations, especially during image preprocessing.

Werkzeug: Provides utilities for file handling, including secure file names.

Main Components:

App Configuration: Setup involves specifying upload folders and paths to pre-trained models for Alzheimer's and Brain Tumors.

Image Preprocessing: Uploaded images are preprocessed to match the input requirements of the models, including resizing and array conversion.

Prediction Endpoints: The Flask app defines endpoints for image uploads and invokes the appropriate model based on the analysis type requested by the user.

Frontend: HTML, CSS, and JavaScript

Key Features:

Navigation and Layout: A modern design with a navigation bar, hero image, and tabbed sections for different analyses.

Dynamic Content with JavaScript: Utilizes jQuery for handling form submissions, making asynchronous requests to the Flask backend, and dynamically updating the UI with analysis results.

Image Preview: Before analysis, users can preview the uploaded image, enhancing the interactive experience.

Styling and Responsiveness: The application is styled with CSS for a professional look and is responsive to different screen sizes, ensuring usability across devices.

Image Analysis Models

Alzheimer's Disease Model: A CNN trained to classify stages of Alzheimer's from MRI scans.

Brain Tumor Model: Another CNN capable of determining the presence of a brain tumor from an MRI scan.

JavaScript Functions:

Tab Navigation: For switching between analysis types, enhancing the user interface by organizing content into separate sections.

Image Analysis Requests: Functions for handling form data, making AJAX requests to the server, and displaying results or errors effectively.

Image Preview: Allows users to see an image preview before submitting for analysis, improving the user experience.

Deployment and Usage

Deployed as a Flask application, users can interact with the MRI Brain Image Analyzer through a web browser. After uploading an MRI scan, the system processes the image using the selected analysis model and returns the prediction, displayed neatly on the web page.

REFERENCES: -

1. Scikit-learn: Machine Learning in Python. (n.d.). Retrieved March 21, 2024, from https://scikit-learn.org/stable/index.html

2. Seaborn: Statistical Data Visualization. (n.d.). Retrieved March 21, 2024, from https://seaborn.pydata.org/

3. TensorFlow. (n.d.). TensorFlow. Retrieved March 21, 2024, from https://www.tensorflow.org/

4. TensorFlow Documentation. (n.d.). Retrieved March 21, 2024, from https://www.tensorflow.org/api_docs