Detecting Alzheimer's via MRI Images Using CNNs and QCNNs

Georgio Feghali, Eric Wang, Jason Wang, Mariana Paco Mendivil, Nicholas Nguyen

July 31, 2024

1 Introduction

This project aims to develop a multi-class classifier that predicts the presence and stage of Alzheimer's disease using MRI images. Utilizing a comprehensive dataset of approximately 5,000 images categorized into Mild Demented, Moderate Demented, Non Demented, and Very Mild Demented, we will create a traditional CNN multi-class classifier and, if time permits, a quantum-based (QCNN) classifier to compare accuracy and training speed. Additionally, we plan to develop an intuitive user interface using Streamlit to facilitate easy interaction with the model, enhancing accessibility and usability for potential users.

Developing a highly accurate predictive model for detecting Alzheimer's disease using MRI images is crucial for early diagnosis and treatment. Alzheimer's disease is a progressive neurodegenerative disorder that affects millions worldwide. Early detection can significantly improve the quality of life for patients by enabling timely interventions and better management of symptoms. By leveraging advanced machine learning techniques, we aim to contribute to the broader impact of improving diagnostic tools in the medical field, ultimately aiding healthcare professionals in making more informed decisions.

2 Methods

2.1 Data Exploration

1. **Image Size**

   We checked the image sizes across the dataset and confirmed that all images in both the training and testing sets have a uniform size of 176x208 pixels.

2. **Image Count Per Class**

   We assessed the distribution of images across the four classes (Mild Demented, Moderate Demented, Non-Demented, and Very Mild Demented) in both the training and testing sets.

3. **Blurriness Check**

We checked for image blurriness with the code below.

```
def check_image_blurriness(data_folder, threshold=100):
    blurry_images = []
    clear_thresholds = []

    folders = os.listdir(data_folder)

    for folder in folders:
        folder_path = os.path.join(data_folder, folder)
        for filename in os.listdir(folder_path):
            image_path = os.path.join(folder_path, filename)
            image = cv2.imread(image_path)

            # Convert to grayscale
            gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

            # Compute the Laplacian of the image and then the variance
            laplacian_var = cv2.Laplacian(gray, cv2.CV_64F).var()

            # Determine if the image is blurry based on the threshold
            if laplacian_var < threshold:
                blurry_images.append(image_path)
                print(f"Blurry image found in {data_folder}: {image_path} (Variance of Laplacian: {laplacian_var})")
            else:
                clear_thresholds.append(laplacian_var)

    # Calculate average clear threshold for non-blurry images
    if clear_thresholds:
        average_threshold = sum(clear_thresholds) / len(clear_thresholds)
        print(f"Average threshold for non-blurry images in {data_folder}: {average_threshold}")
    else:
        average_threshold = None
        print(f"No clear images found in {data_folder}.")

    return blurry_images, average_threshold

# Example usage:
alzheimer_train = "Alzheimer_s Dataset/train"
alzheimer_test = "Alzheimer_s Dataset/test"

train_blurry_images, avg_threshold_train = check_image_blurriness(alzheimer_train)
print()
test_blurry_images, avg_threshold_test = check_image_blurriness(alzheimer_test)

print("\nBlurry images in training set:")
print(train_blurry_images)

print("\nBlurry images in test set:")
print(test_blurry_images)
```

4. **Color Distribution**

We examined the color distribution for each class.

5. **Visualizing Sample Images**

   We plotted examples of images from each class.

2.2 Data Preprocessing

The preprocessing phase is crucial for preparing our MRI image dataset for training our models. This phase involves several essential steps to ensure that the data is in the best possible format for model training.

1. **Addressing Class Imbalance**

   We will be combining all the cases of Alzheimer's into 1 class. Therefore, our datasets will be sorted into 2 classes, Not Demented and Demented.

2. **Image Normalization**

   Normalization is essential for standardizing the pixel values across all images. We will use the following normalization technique:

   Scaling to [0, 1]: All pixel values will be divided by 255 to scale them to the range [0, 1]. This is because pixel values in an 8-bit image range from 0 to 255, and dividing by 255 scales them to the desired range, basically having the same effect as a MinMax normalization. The code for our MinMax normalization is below:

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

# ImageDataGenerator for normalization only (no data augmentation)
train_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

# Load and preprocess images from the directories
train_generator = train_datagen.flow_from_directory(alzheimer_train,
                                                    target_size=(176, 208),
                                                    batch_size=32,
                                                    class_mode='binary')

test_generator = test_datagen.flow_from_directory(alzheimer_test,
                                                  target_size=(176, 208),
                                                  batch_size=32,
                                                  class_mode='binary')
```

2.3 Model 1: Convolutional Neural Network (CNN)

For our first model, we built a convolutional neural network without dropout and data augmentation using TensorFlow and Keras to classify MRI scans into demented and non-demented categories. Our model consisted of several convolutional and max-pooling layers, followed by a fully connected layer. We trained the model with a batch size of 32 for 10 epochs, monitoring its performance using a separate validation set.

1. **Model Architecture:**

   Convolutional Layer 1: 32 filters, kernel size 3x3, activation ReLU

   Max-Pooling Layer 1: pool size 2x2

   Convolutional Layer 2: 64 filters, kernel size 3x3, activation ReLU

   Max-Pooling Layer 2: pool size 2x2

   Convolutional Layer 1: 32 filters, kernel size 3x3, activation ReLU

   Max-Pooling Layer 1: pool size 2x2

   Convolutional Layer 2: 64 filters, kernel size 3x3, activation ReLU

   Max-Pooling Layer 2: pool size 2x2

   Fully Connected Layer: 512 units, activation ReLU

   Output Layer: 1 unit, activation Sigmoid

2. **Model Hyperparameters:**

   Hidden Layers Activation Function: ReLU

   Output Layer Activation Function: Sigmoid

   Loss Function: Binary Cross Entropy

   Optimizer: RMSprop

   Learning Rate: 0.0001

   Metric: Accuracy

3. **Training Configuration:**

   Batch Size: 32

   Epochs: 10

   The code for our first model is below:

```
from tensorflow.keras import layers, models

model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(176, 208, 3))
    ,
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])
model.compile(loss='binary_crossentropy',
              optimizer=tf.keras.optimizers.RMSprop(learning_rate=1e-4),
              metrics=['accuracy'])
history = model.fit(train_generator,
```

```
                    steps_per_epoch=100,
                    epochs=10,
                    validation_data=test_generator,
                    validation_steps=30)
```

2.4 Second Model: Enhanced Convolutional Neural Network (CNN)

For our second model, we implemented several improvements to address overfitting and enhance the overall performance of our CNN. These modifications included data augmentation, dropout layers, and structural changes to the network.

1. **Changes Made from first model:**

   Data Augmentation:

   RandomZoom: Applied to slightly zoom into images.

   RandomFlip: Used to horizontally flip images.

   RandomRotation: Applied to rotate images randomly.

   We implemented data augmentation using the following Keras preprocessing layers: tf.keras.layers.RandomFlip, tf.keras.layers.RandomRotation, and tf.keras.layers.RandomZoom.

   Dropout Layer:

   Dropout Rate: Added a dropout layer with a rate of 0.5 before the dense layer. This technique randomly sets a fraction of the input units to 0 during training.

   CNN Structure Changes:

   Number of Layers: Our second model includes 3 additional layers compared to the first model.

   Preprocessing Layers: Unlike the first model, preprocessing such as normalization is handled within the neural network itself.

   Dropout Layer Addition:

   Incorporated a dropout layer before the dense layer, which was absent in the first model.

   The code for our second model is below.

```
model = Sequential([
   data_augmentation,
  layers.Rescaling(1./255),
  layers.Conv2D(32, (3, 3), activation='relu'),
  layers.MaxPooling2D((2, 2)),
  layers.Conv2D(64, (3, 3), activation='relu'),
  layers.MaxPooling2D((2, 2)),
  layers.Conv2D(128, (3, 3), activation='relu'),
  layers.MaxPooling2D((2, 2)),
```

```
  layers.Conv2D(128, (3, 3), activation='relu'),
  layers.MaxPooling2D((2, 2)),
  layers.Flatten(),
  layers.Dropout(0.5),
  layers.Dense(512, activation='relu'),
  layers.Dense(1, activation='sigmoid')
])
```

2.5 Final Model - Hyperparameter tuning

In our final model, we focused on optimizing the performance of our convolutional neural network (CNN) by tuning several key hyperparameters.

1. **Hyperparameters Tuned:**

   Learning Rate, Optimizer, Activation Function for the Convolutional Layers, Activation Function for the Dense Layers

   We utilized random search to explore different combinations of these hyperparameters and identified the optimal settings:

   Optimal Learning Rate: 0.0001

   Optimal Optimizer: Adam

   Optimal Activation Function for Convolutional Layers: Tanh

   Optimal Activation Function for Dense Layers: ReLU

2. **Training Configuration**

   With these tuned hyperparameters, we retrained the model for 100 epochs to ensure that the network could effectively learn from the data without overfitting. The code to find our best hyperparameter tuning model is below, as well as the code for our best hyperparameter tuning model.

```
def build_model(hp):
    model = Sequential([
        data_augmentation,
        layers.Rescaling(1./255),
        layers.Conv2D(32, (3, 3), activation=hp.Choice('conv_activation',
    values=['relu', 'tanh', 'sigmoid'])),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation=hp.Choice('conv_activation',
    values=['relu', 'tanh', 'sigmoid'])),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(128, (3, 3), activation=hp.Choice('conv_activation',
    values=['relu', 'tanh', 'sigmoid'])),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(128, (3, 3), activation=hp.Choice('conv_activation',
    values=['relu', 'tanh', 'sigmoid'])),
        layers.MaxPooling2D((2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(512, activation=hp.Choice('dense_activation', values
    =['relu', 'tanh', 'sigmoid'])),
        layers.Dense(1, activation='sigmoid')
```

```
    ])

    # Choose the optimizer
    optimizer = hp.Choice('optimizer', values=['adam', 'rmsprop', 'sgd'])

    if optimizer == 'adam':
        opt = tf.keras.optimizers.Adam(learning_rate=hp.Choice('
     learning_rate', values=[1e-2, 1e-3, 1e-4]))
    elif optimizer == 'rmsprop':
        opt = tf.keras.optimizers.RMSprop(learning_rate=hp.Choice('
     learning_rate', values=[1e-2, 1e-3, 1e-4]))
    elif optimizer == 'sgd':
        opt = tf.keras.optimizers.SGD(learning_rate=hp.Choice('
     learning_rate', values=[1e-2, 1e-3, 1e-4]))

    model.compile(optimizer=opt,
                  loss='binary_crossentropy',
                  metrics=['accuracy'])

    return model
```

```
def build_best_model(hp):
    model = Sequential([
        data_augmentation,
        layers.Rescaling(1./255),
        layers.Conv2D(32, (3, 3), activation=hp.get('conv_activation')),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation=hp.get('conv_activation')),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(128, (3, 3), activation=hp.get('conv_activation')),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(128, (3, 3), activation=hp.get('conv_activation')),
        layers.MaxPooling2D((2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(512, activation=hp.get('dense_activation')),
        layers.Dense(1, activation='sigmoid')
    ])

    if hp.get('optimizer') == 'adam':
        opt = tf.keras.optimizers.Adam(learning_rate=hp.get('learning_rate
     '))
    elif hp.get('optimizer') == 'rmsprop':
        opt = tf.keras.optimizers.RMSprop(learning_rate=hp.get('
     learning_rate'))
    elif hp.get('optimizer') == 'sgd':
        opt = tf.keras.optimizers.SGD(learning_rate=hp.get('learning_rate')
     )

    model.compile(optimizer=opt,
                  loss='binary_crossentropy',
                  metrics=['accuracy'])

    return model

# Build the model with the best hyperparameters
best_model = build_best_model(best_hps)
```

3 Results

3.1 Results of Data Exploration:

Below are the histograms showing the number of samples per class for the training and testing sets.

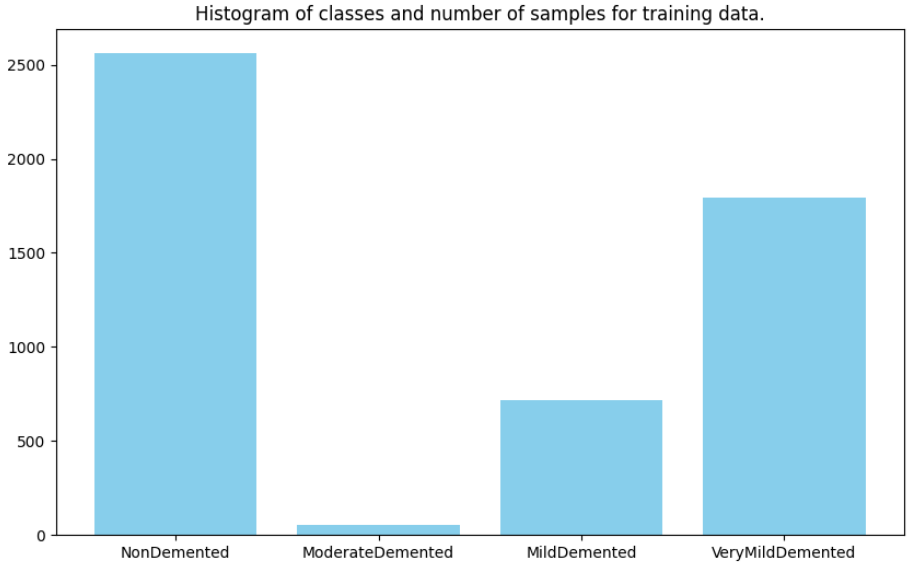


Figure 1: Distribution of samples per training class

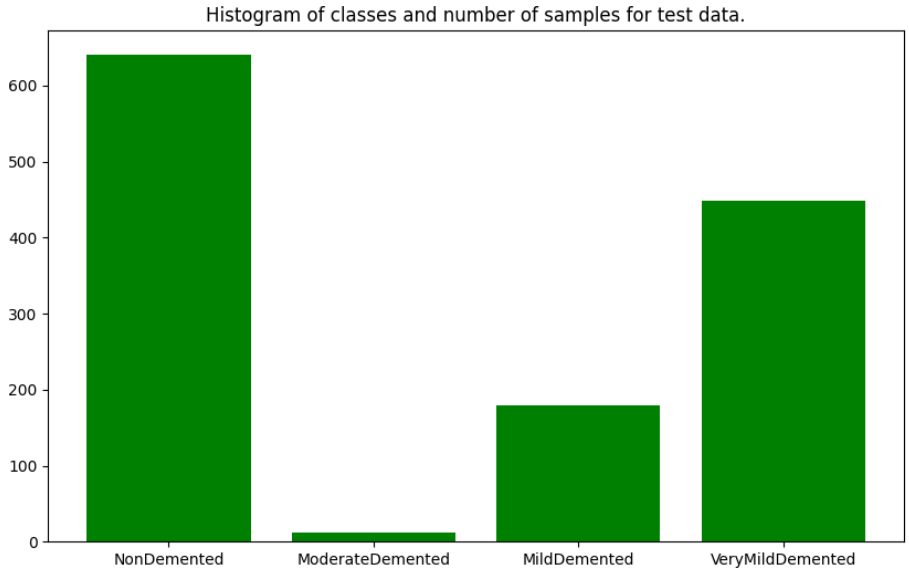


Figure 2: Distribution of samples per testing class

More on the histograms of the classes will be considered in the discussion section.

From the blurriness code from the methods section, we found that all images were of consistent quality with no outliers in terms of blurriness.

We plotted the color distribution and confirmed that the grayscale intensity levels were consistent across all classes for both the training and testing sections.

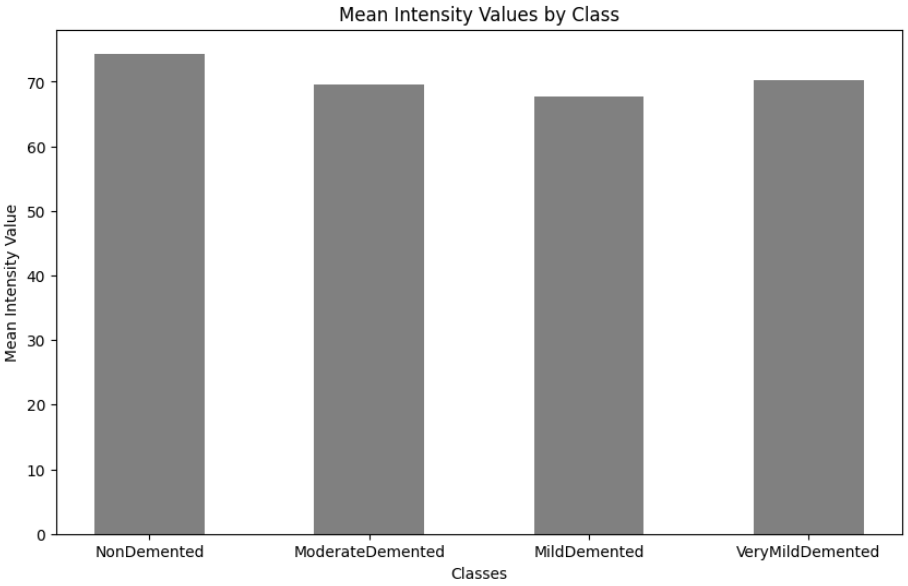


Figure 3: Grayscale intensity distribution across training classes

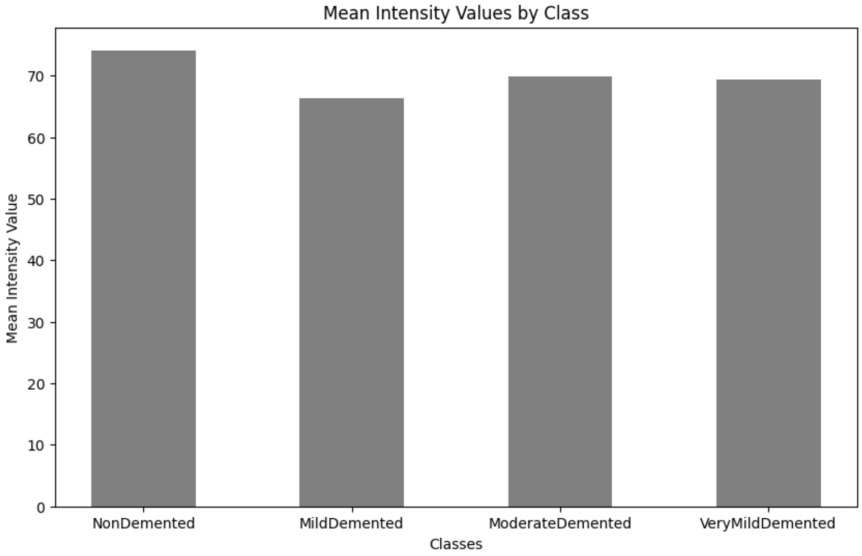


Figure 4: Grayscale intensity distribution across testing classes

Below are the sample image results from each class.

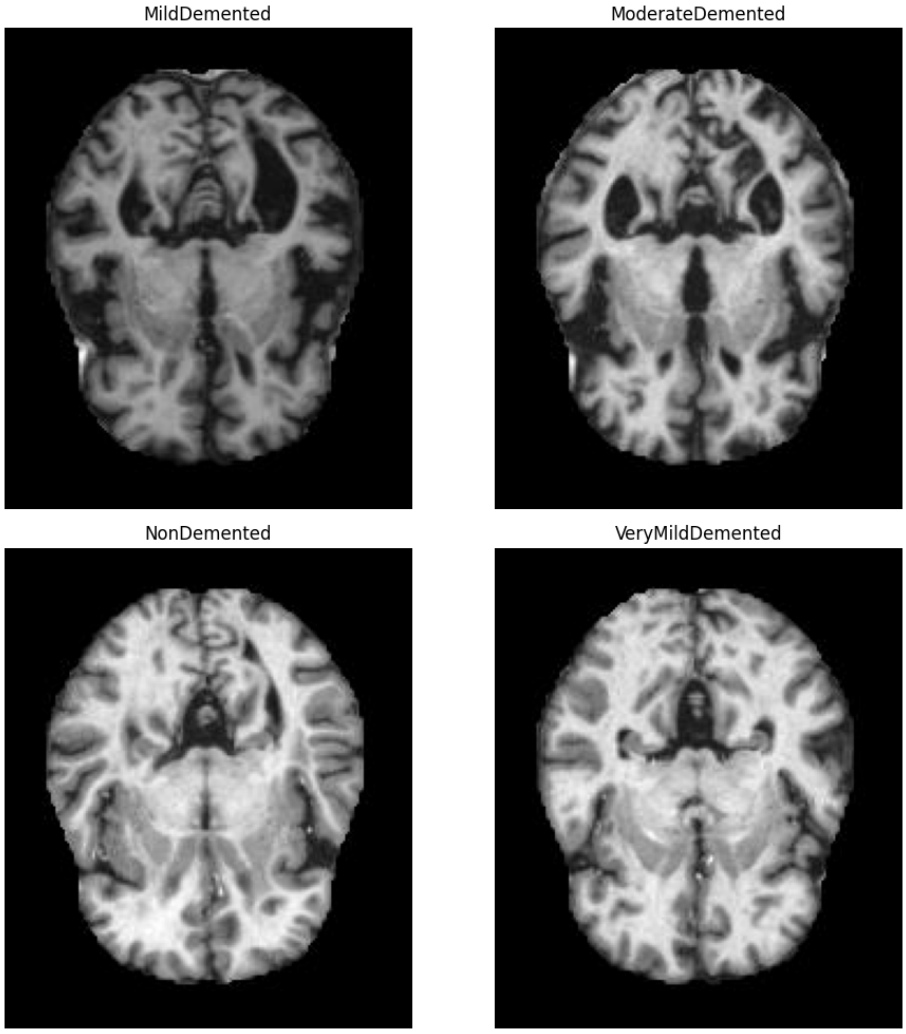


Figure 5: Sample images from each class

3.2 Results of Preprocessing:

When we combine the Mild Demented, Moderate Demented, and Very Mild Demented Classes into one Demented class, we see that the Not Demented class contains 3200 images spanning both the test and training sets, and the Demented class will have the same amount, 3200 images spanning both the test and training sets. More Specifically, We have 639 Demented cases and 640 Non Demented cases for our test data and 2561 Demented cases and 2560 Non Demented cases for our training data. From our normalization, each pixel is now in the range of [0, 1].

3.3 Results from the First Model:

After training, we evaluated the model on both the training and test datasets to measure its accuracy and loss, and visualized the training progress by plotting the training and validation loss and accuracy over the epochs.

After 10 epochs:
Train Loss: 0.23
Train Accuracy: 0.91
Validation Loss: 0.68
Validation Accuracy: 0.65

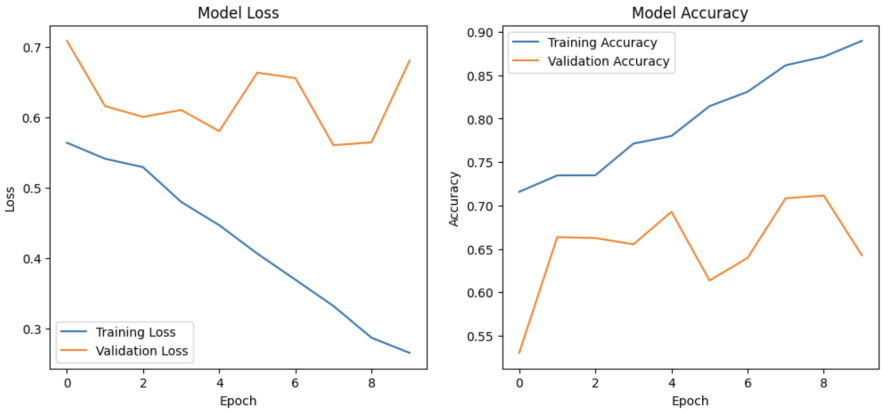
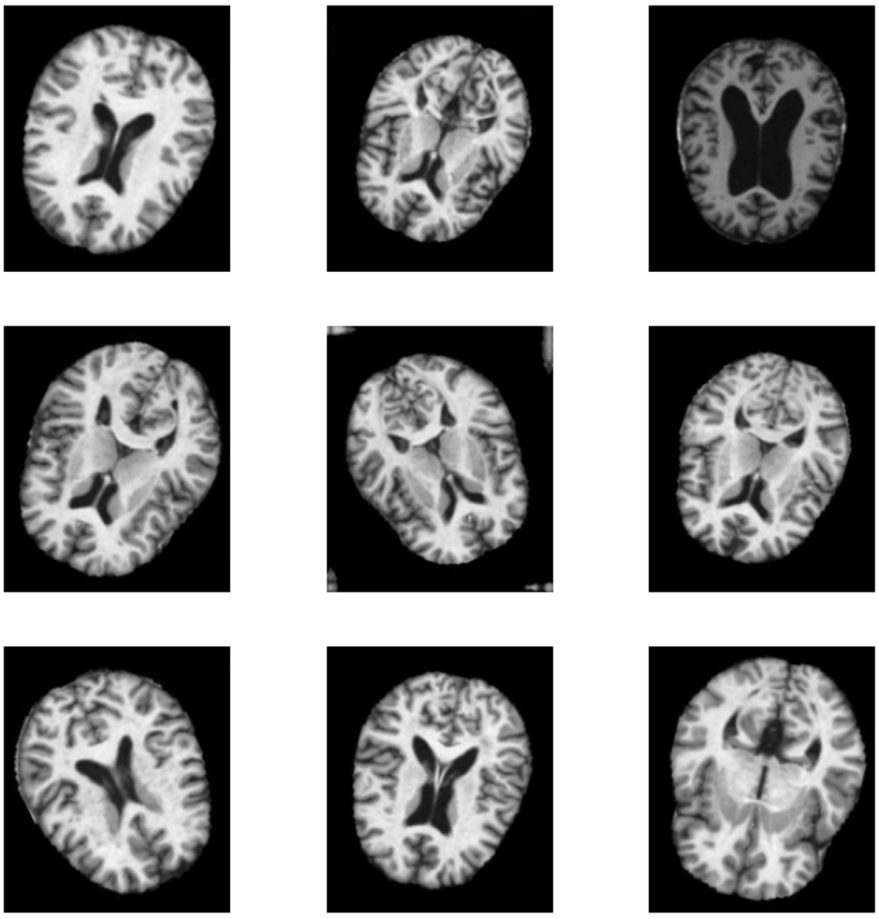


Figure 6: First model loss and accuracy

3.4 Results from the Second Model:

1. **Results from data augmentation**



2. **Results from second model:**

   The accuracy after 100 epochs for both training and validation datasets was around 92 percent.

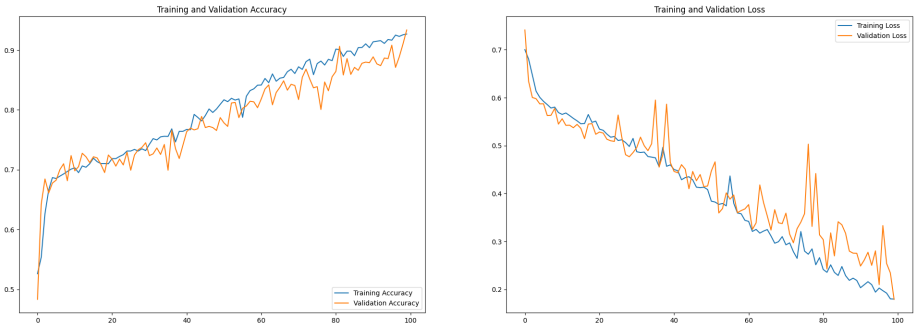


Figure 7: Second model loss and accuracy

	precision	recall	f1-score	support
AllDemented	0.72	0.82	0.77	639
NonDemented	0.79	0.69	0.74	640
accuracy			0.75	1279
macro avg	0.76	0.75	0.75	1279
weighted avg	0.76	0.75	0.75	1279

Figure 8: Second model classification score

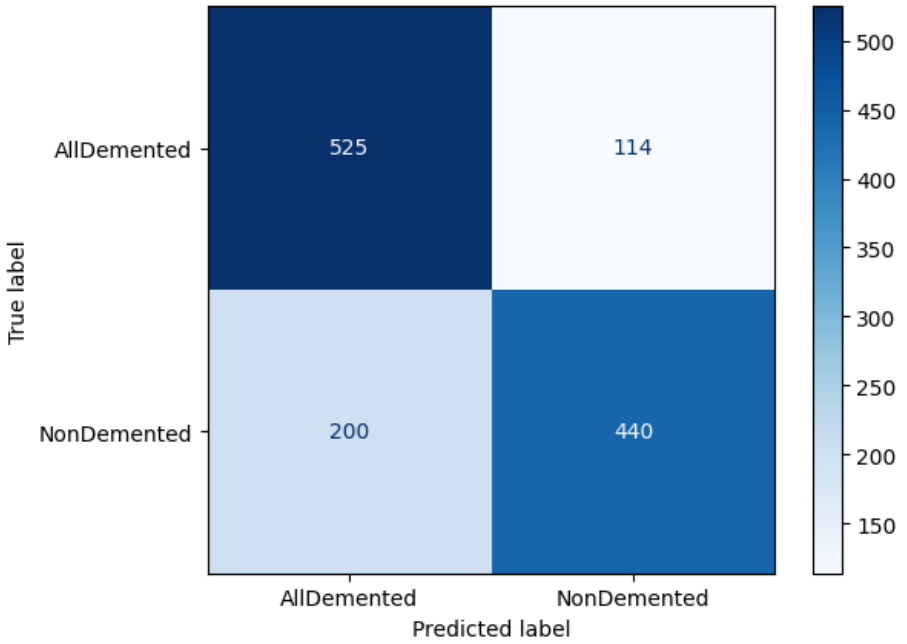


Figure 9: Second model confusion matrix

3.5 Results from the Final Model:

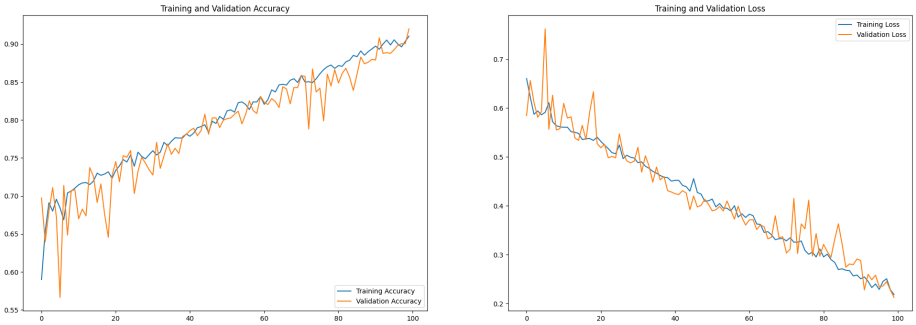


Figure 10: Final model loss and accuracy

	precision	recall	f1-score	support
AllDemented	0.72	0.82	0.77	639
NonDemented	0.79	0.69	0.74	640
accuracy			0.75	1279
macro avg	0.76	0.75	0.75	1279
weighted avg	0.76	0.75	0.75	1279

Figure 11: Final model classification score

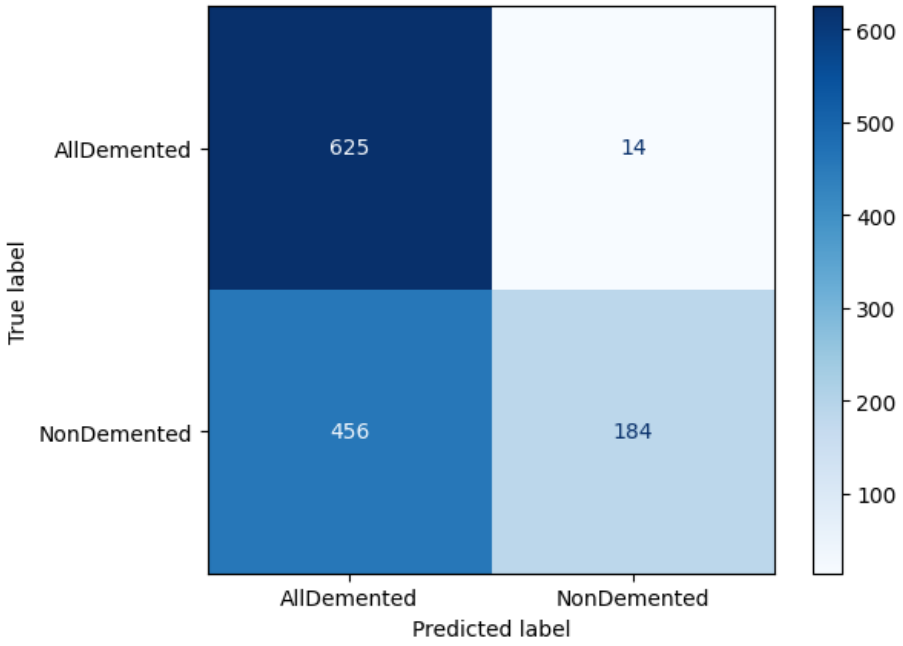


Figure 12: Final model confusion matrix

1. **Model Performance Metrics:**

   Training Accuracy: around 92 percent

   Validation Accuracy: around 92 percent

   Final Accuracy on Test Data: 63 percent

4 Discussion

4.1 Data Exploration

The uniform image sizes simplify the preprocessing steps, ensuring consistent model input dimensions. As a result, no additional cropping is required since all images are already the same size (176x208 pixels). However, from this step, we learned that normalization must be performed in order to standardize the images so that model can perform learning optimally.

We counted the distribution of images across the four classes (Mild Demented, Moderate Demented, Non-Demented, and Very Mild Demented) in both the training and testing sets to understand the class imbalance in our dataset. As illustrated in the histograms in the results section, there is a significant imbalance in the dataset. The "Non-Demented" class has the highest number of samples, followed by "Very Mild Demented", "Mild Demented", and "Moderate Demented". This imbalance can lead to a biased model that performs well on the majority class but poorly on the underrepresented classes, potentially reducing the overall accuracy and effectiveness of the model. This means we need to figure out a way to balance the classes in the preprocessing step.

To ensure that our dataset consists of high-quality images, we checked for image blurriness. This step was crucial because blurry images could negatively impact the model's ability to learn and make accurate predictions. Our analysis confirmed that all images were of consistent quality with no significant outliers in terms of blurriness (code available in the methods section). This consistency guarantees that our model is trained on clear and precise images, enhancing its accuracy and reliability.

Since MRI images are typically grayscale, we examined the grayscale intensity distribution for each class to verify uniformity. This step is crucial to ensure that the intensity levels are consistent across all classes, which helps in understanding the inherent differences in image characteristics for different stages of Alzheimer's disease. We see that from the graphs in the results section that the mean intensity values by class is all within the 65-75 range, so the intensity levels are pretty consistent.

Additionally, we plotted examples of images from each class to get a better visual understanding of the dataset. These visualizations provided insights into the subtle differences and similarities in MRI images for each stage of Alzheimer's, which is essential for effective model training.

4.2 Data preprocessing

Given the significant class imbalance in our dataset, we combined all cases of Alzheimer's into one class. Therefore, our datasets were sorted into two classes: Not Demented and Demented. As stated in the results section, this step addressed the class imbalance issue, ensuring that the Not Demented class contained 3200 images spanning both the test and training sets, and the De-

mented class had the same amount, 3200 images spanning both the test and training sets. This rebalancing is crucial to prevent the model from becoming biased towards the majority class, which could significantly impact its ability to accurately identify cases of Alzheimer's disease.

Normalization is essential for standardizing the pixel values across all images. When the input data is normalized, it helps the model converge faster during training. This is because the gradients computed during the optimization process become more stable and consistent, allowing for more efficient and effective learning. Also, normalization ensures that the input data is consistent across different images and batches. This consistency is crucial for the model to learn and generalize patterns effectively from the data.

4.3 First Model

The first model exhibited high training accuracy but considerably lower validation accuracy, indicating overfitting. Overfitting occurs when the model learns to perform very well on the training data by memorizing it, but fails to generalize to unseen validation or test data. The training accuracy was 91 percent, whereas the validation accuracy was only 65 percent. From the graphs, we can see that the training loss consistently decreased, but the validation loss did not decrease at the same rate, and it even remained relatively high, which is a sign of overfitting. This discrepancy suggests that the model was capturing noise and specific patterns from the training data that were not present in the validation data, leading to poor generalization.

Plan for Improvement

To address the overfitting issue and enhance the model's generalization capabilities, we plan to implement the following improvements:

1. **Data Augmentation:**

   Data augmentation techniques such as rotation, flipping, zooming, and shifting will artificially expand the training dataset and introduce variability, making the model more robust to different input scenarios. By augmenting the data, the model is exposed to a wider range of scenarios and variations, which prevents it from memorizing specific patterns in the training set and helps it generalize better to unseen data.

2. **Incorporating Dropout Layers:**

   Adding dropout layers will help prevent the model from becoming too reliant on specific neurons. Dropout randomly drops a set of neurons during training, forcing the network to learn more general and robust features. This helps in reducing overfitting and improving the model's ability to generalize.

3. **Hyperparameter Tuning:**

   Conducting hyperparameter tuning to find the optimal values for the learning rate, batch size, number of epochs, and other parameters will

help improve the model's performance. Fine-tuning hyperparameters ensures that the model is trained with the best possible settings, which can enhance its performance and generalization capabilities. Proper tuning can lead to better convergence and reduced overfitting.

4. **Using Early Stopping:**

   Implementing early stopping will halt the training process once the validation performance stops improving. This will prevent the model from continuing to train on the noise in the training data after reaching the optimal point of generalization. Lastly, early stopping avoids overfitting and ensures that the model performs well on validation and test data.

5. **Exploring Advanced Architectures:**

   If time permits, we will explore more advanced architectures, such as quantum-based CNNs (QCNN), to compare accuracy and training speed. By applying these techniques, we aim to build a more robust and generalizable model that performs well on both training and validation datasets.

   Therefore, the initial model, while effective in achieving a reasonable classification accuracy, showed clear signs of overfitting. Our next steps involve implementing the improvements outlined above to enhance the model's performance. Specifically, we will focus on data augmentation, dropout layers, hyperparameter tuning, and early stopping to mitigate overfitting and achieve better generalization. Additionally, we will consider exploring more advanced architectures to further improve the model's capabilities.

4.4 Second Model

We implemented RandomZoom so that it added variation to the dataset. We implemented RandomFlip, helping the model learn to recognize features irrespective of their orientation. We added RandomRotation to further increasing the diversity of training examples. These augmentations were designed to generate additional training examples from our existing data by applying transformations that create realistic variations of the images. This process helps expose the model to a wider range of data and improves its ability to generalize. We added the dropout layer to help prevent the model from becoming too reliant on specific neurons and reduces overfitting. We increased the number of layers in the second model to increase its capacity to learn complex features.

The improvements we made significantly enhanced the model's performance.

1. **Training and Validation Performance:**

   The training accuracy improved linearly, and the validation accuracy showed consistent growth. Similarly, the training and validation errors decreased linearly, indicating that the model is now better at generalizing without overfitting. The model has reached the optimal region on the fitting graph. As stated before in the results section, the accuracy after 100 epochs for

both training and validation datasets was around 92 percent, which is extremely good.

2. **Test Data Performance:**

   The model achieved approximately 75 percent accuracy on the test data, marking a 50 percent improvement from the base model, which had an accuracy of 50 percent. This demonstrates the effectiveness of the applied improvements. The recall was 0.82, showing a substantial improvement over the first model. This metric is crucial for detecting Alzheimer's cases, as it minimizes false negatives and improves the model's ability to identify patients with the disease.

   While the second model showed significant improvements, particularly in recall, there is still room for enhancement. For our final model, we aim to further increase both the test accuracy and recall for the "All Demented" class through hyperparameter tuning. This will help optimize the model's performance and ensure that it effectively balances accuracy and recall for critical medical applications.

4.5 Final Model: Hyperparameter Tuning

The goal of the hyperparameter tuning model is to improve the model's ability to generalize and reduce the risk of overfitting observed in earlier versions.

After training with the optimized hyperparameters, the model exhibited controlled learning and no significant overfitting, as observed in the training and validation curves in the results section. The accuracy plateaued at around 92 percent for both the training and validation datasets, similar to the second model.

In addition to the high accuracy, the most notable improvement was observed in the recall for the "All Demented" class, which reached 0.98. This indicates that the model successfully identifies Alzheimer's cases in 98 percent of instances, which is a substantial improvement over previous models.

From the confusion matrix in the results section, The model demonstrated a trade-off, where it predicted "All Demented" more frequently than "NonDemented", leading to a lower overall accuracy but a significantly higher recall for dementia detection.

While the final model did not achieve a significant improvement in overall accuracy, the increase in recall for detecting Alzheimer's disease is critical for this application. The model's ability to minimize false negatives (cases where dementia is missed) is particularly valuable in a medical context, where early detection is crucial. This model prioritizes recall over accuracy, which is a suitable trade-off given the application.

5 Bonus: Quantum Approach

We know the matrix computation for classic computer is expensive, so we tried to find the feasibility of training the neural network as a quantum approach.

5.1 Feature Maps:

Based on our research, the industry latest quantum computer can handle around 500 qubits at a time. However, when training neural network, we need to treat each pixel as a feature for the input. We have over 30,000 pixels for each picture in our dataset, it is impossible to convert all the features into quantum states. We need to find a way to reduce the quantity of features we have. The most straight forward method is downscaling. It can reduce the number of features to the maximum number that we can handle. However, this approach could loss a lot of information.

5.2 Improvement Idea:

In order to increase the accuracy of our model and reduce the impact of the information loss, we have an idea to improve the model. Theoretically, we can cut an image into smaller parts with the same size that could handled by a quantum computer, then we train these small parts first and check the accuracy to see if the patch of the image is useful for us. Then we use the accuracy and the mean value of that small patch as new features for next round training. After several rounds of iteration, we can train a large image with multiple smaller parts. There is no rigorous prove for this idea, and this approach could also cause some issue after data augmentation. But it seems doable with the dataset we chose.

5.3 Data Preprocessing:

Basically, the data preprocessing of the quantum part is very similar to what we do for the traditional machine learning. We just ensure our data are normalized, and downscaled all the images to make sure we are able to convert each feature into a limited number of superstates. We also separated the whole dataset into the training set, validation set and a test set.

5.4 Create Quantum Circuit:

We need to design the quantum circuit based on our model in quantum computation. It is a sequence of quantum gates that represents the transformation on the quantum states. In our model, we downscaled the image into 256 pixels. So we need to initialize a 16 x 16 grid of qubits which represents 256 qubits. In order to translate the image data into quantum states, we applied a X gate, which means NOT, to each state where the pixel value is non-zero. The code for the translation is below.

```
def create_circuit_from_image(encoded_image):
  """
  Returns a circuit for given encoded image

  Parameters:
  encoded_image (array): Encoded Image

  Returns:
  circuit (cirq.Circuit object): cirq circuit
  """
  qubits = cirq.GridQubit.rect(16,16)
  circuit = cirq.Circuit()
  for i, pixel in enumerate(encoded_image):
    if pixel:
      circuit.append(cirq.X(qubits[i]))
  return circuit
```

5.5 Quantum Neural Network:

We used cirq and tensorflow_quantum libraries to help us build the QNN. The basic idea is that we apply a Hardamard gate to each quantum states to convert them into super positions. Then we add some layers like ZZ or XX. After that, we add Hardamard gates one more time, and measure the results that a quantum computer calculated. Then we do the same as a traditional quantum computer does. We compile our model with our ideal loss function and optimizer, and fit our data with specific batch size and epochs. But for this time, we do not need to care about the matrix calculation and activation functions.

6 Conclusion

This project didn't necessarily require a quantum neural network, but our curiosity drove us to explore its potential and how it could improve our models. But reflecting on the project, there are several areas where improvements could have been made and directions for future work:

6.1 Advanced Preprocessing:

Employing more sophisticated preprocessing techniques could have significantly enhanced the quality of the MRI images fed into the model, potentially leading to better feature extraction and improved classification accuracy. One such technique is cropping the images to show only the brain. This process involves removing irrelevant parts of the image, such as the skull and surrounding tissues, to focus solely on the brain region. By doing so, the model can concentrate on the critical areas that are most relevant to diagnosing Alzheimer's disease, reducing noise and improving the signal-to-noise ratio.

6.2 Hyperparameter Tuning:

More extensive hyperparameter tuning using techniques such as grid search or Bayesian optimization could have potentially led to better model performance. Exploring different architectures and depths of the CNN might have revealed an optimal configuration for this specific dataset.

6.3 Quantum-Based Classifier:

Although time constraints may have limited the implementation of the QCNN, exploring this cutting-edge approach could yield significant insights into the potential benefits of quantum computing in medical image analysis. Future work could focus on fully developing and comparing the QCNN with the traditional CNN.

6.4 User Interface Enhancements:

While the initial development of the Streamlit interface is a great start, future work could involve refining the user experience, adding functionalities such as real-time prediction, and providing detailed reports on each prediction to aid clinicians in decision-making.

Overall, the project was a resounding success, characterized by excellent team collaboration, thorough communication, and achieving the anticipated results. The collective effort and dedication of the team members facilitated smooth progress and allowed us to overcome challenges efficiently. The positive outcomes of this project underscore the importance of teamwork and a shared commitment to our goals.

7 Statement of Collaboration

1. **Georgio Feghali (Leader/Programmer/Writer):** Worked on the code for Pre-processing, the Second Model, and Hyperparameter Tuning. Also wrote up the ReadMe files on GitHub.
2. **Mariana Paco Mendivil (Programmer):** Worked on the code for Data Exploration, Pre-processing, the First Model, and the User Interface.
3. **Nicholas Nguyen (Programmer):** Worked on the code for Data Exploration, Pre-processing, and the First Model. Scheduled team meetings and deadlines.
4. **Eric Wang (Writer):** Wrote up the entire Written Report and gave feedback to programmers on plots/images. Checked over everyone's code, suggested possible ideas/improvements, and made sure everything was set to submit.
5. **Jason Wang (Programmer):** Worked on the code for Data Exploration, Pre-processing, the First Model, and Quantum neural networks.