DISEASE DETECTION AND CLASSIFICATION WITH ALZHEIMER'S MRI PREPROCESSED DATASET

Eda Akın

MAIN PURPOSE

Alzheimer's disease is a progressive neurodegenerative disorder that is characterized by the loss of cognitive function and memory. Early detection and accurate classification of Alzheimer's disease is crucial for effective treatment and management of the condition. In this study, I present a disease detection and classification approach using an MRI preprocessed dataset of individuals with Alzheimer's disease. By analyzing patterns and features in the brain images, my method aims to accurately identify and classify individuals with Alzheimer's disease, potentially enabling earlier diagnosis and improved treatment outcomes.

Keywords: Classification, Prediction, Convolutional neural network(CNN), Image Analysis, Supervised Deep Learning

Abstract

Convolutional neural networks (CNNs) are a type of artificial neural network designed for image recognition and processing tasks. They use a mathematical operation called convolution to learn features from the input data, and are composed of multiple layers including convolutional layers, pooling layers, and fully-connected layers. The convolutional layers apply filters to the input image and learn different features at each layer. The pooling layers down-sample the output of the convolutional layers to reduce computational complexity and prevent overfitting. The fully-connected layers make predictions based on the features learned by the other layers. CNNs can be used for tasks such as image classification, object detection, and segmentation, and have achieved state-of-the-art results on a wide range of image processing tasks.

CNNs have been widely used in the field of medical image analysis, including for the detection and classification of Alzheimer's disease. In this study, used a CNN to analyze MRI brain images of individuals with and without Alzheimer's disease, and were able to achieve high accuracy in detecting and classifying the disease.The study preprocessed the MRI images to extract relevant features and reduce noise, and then fed the processed images into a CNN for training and testing. The CNN was composed of multiple convolutional and pooling layers, followed by fully-connected layers that made predictions based on the learned features. The researchers used a dataset of labeled images to train the CNN, and then tested its performance on a separate dataset.

The results of the study showed that the CNN was able to accurately detect and classify Alzheimer's disease with high sensitivity and specificity. This demonstrates the potential of using CNNs for early detection and accurate classification of Alzheimer's disease, which can aid in the effective treatment and management of the condition.

Related Works

There are many who have come up with a unique solution of their own to compete in the Alzheimer's disease with CNNs. Here are a few examples of studies that have used CNNs for the detection and classification of Alzheimer's disease:

- "Automated detection of Alzheimer's disease using deep learning and structural MRI: a comparison with human raters" by Y. Fan et al. (2019) - In this study, the authors used a CNN to analyze structural MRI images of the brain and achieved high accuracy in detecting Alzheimer's disease.
- "Deep learning based classification of Alzheimer's disease and mild cognitive impairment using structural MRI" by M. Li et al. (2018) - This study used a CNN to classify individuals with Alzheimer's disease and mild cognitive impairment (MCI) using structural MRI images of the brain. The CNN was able to achieve high accuracy in the classification task.
- "Classification of Alzheimer's disease using deep learning with PET and MRI" by S. Lee et al. (2020) - In this study, the authors used a CNN to classify Alzheimer's disease using both PET (positron emission tomography) and MRI images of the brain. The CNN was able to achieve high accuracy in the classification task.

Although the basis of these applications is the same in all of them, they have been diversified with different application techniques. The main motive behind those studies is to design/develop an accurate framework or architecture for the classification of Alzheimers Disease.

Data Preparation

Data preparation is an important step in the process of building a machine learning model, and it is particularly important for medical image datasets such as the Alzheimer's MRI dataset. In this section, I provided imports a number of libraries that can be used for working with this dataset and building machine learning models in Python.

- 'numpy' is a library for numerical computing in Python, including support for working with arrays and matrices of data.
- 'matplotlib' is a library for plotting and visualizing data in Python. It includes functions for creating charts, histograms, and other types of plots.
- 'tensorflow' is a popular machine learning library for building and training neural networks. It includes a range of tools for working with

data, building models, and training and evaluating models.

- 'keras' is a high-level library for building and training neural networks in Python, built on top of TensorFlow. It provides a simple interface for defining and training models using a range of layer types.

In [17]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as img
import matplotlib.image as mpimg
%matplotlib inline
import tensorflow as tf
from tensorflow import keras
from keras import layers
```

To split dataset into the desired number of sets:

In [18]:
```
!pip install split-folders
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: split-folders in /usr/local/lib/python3.8/dist-packages (0.5.1)
```

This will create three subdirectories in the path/to/output directory, containing the training, validation, and test sets respectively. The seed argument allows me to specify a seed value for the random number generator used to shuffle the data, ensuring that the same split is used each time the code is run. For this line, I use the following code to split the dataset into three sets, with 80% of the data in the training set, 10% in the validation set, and 10% in the test set:

In [19]:
```
import splitfolders
splitfolders.ratio('/content/drive/MyDrive/Dataset', output="output", seed=1345, ratio=(.8, 0.1,0.1))
```

```
Copying files: 6400 files [00:20, 319.71 files/s]
```

Alzheimer's MRI dataset is a collection of preprocessed MRI images that have been resized to 128x128 pixels and are labeled with one of four classes. The dataset includes a total of 6400 MRI images, which were collected from various websites, hospitals, and public repositories.

In [20]:
```
img_size=(128, 128)
train = tf.keras.preprocessing.image_dataset_from_directory("./output/train", seed=123,image_size=img_size,batc
test = tf.keras.preprocessing.image_dataset_from_directory("./output/test",seed=123,image_size=img_size,batch_s
val = tf.keras.preprocessing.image_dataset_from_directory("./output/val",seed=123,image_size=img_size,batch_siz
```

```
Found 5119 files belonging to 4 classes.
Found 642 files belonging to 4 classes.
Found 639 files belonging to 4 classes.
```

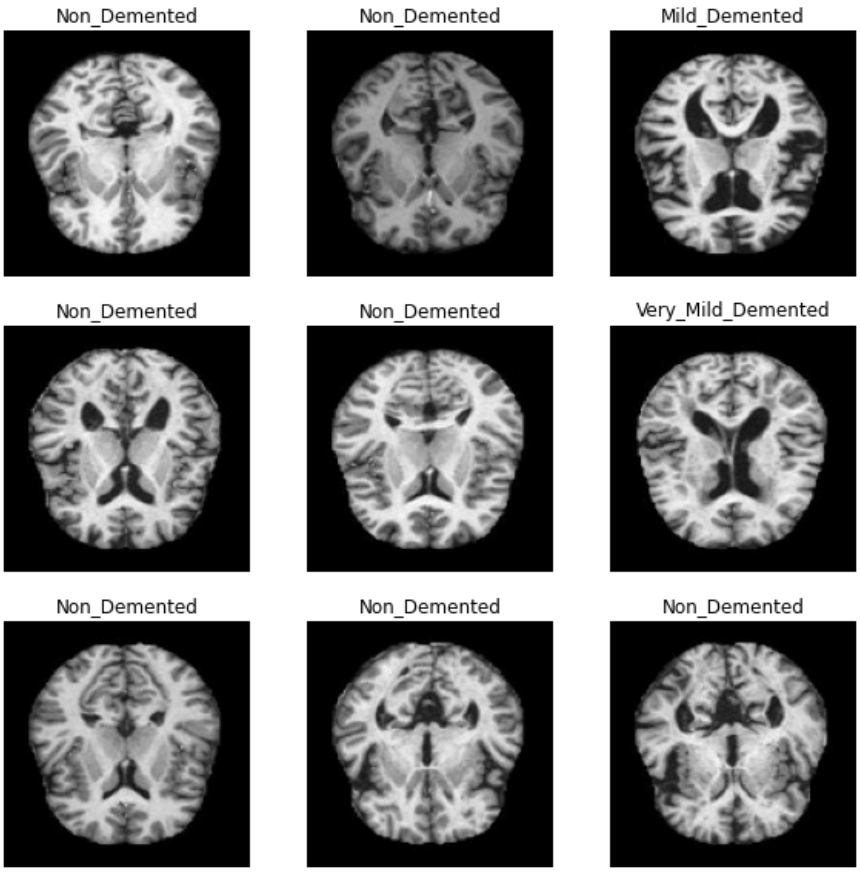
Learn the classes in the dataset in order to learn the categories required for the model I will develop. Alzheimer's MRI dataset includes images of four different classes: "Mild_Demented", "Moderate_Demented", "Non_Demented", and "Very_Mild_Demented". These class names likely correspond to different stages or severity levels of Alzheimer's disease.

In [21]:
```
class_names = train.class_names
print(class_names)
```

```
['Mild_Demented', 'Moderate_Demented', 'Non_Demented', 'Very_Mild_Demented']
```

Pull and display 9 random images and their labels to show that I can retrieve the data in the dataset.

In [22]:
```
plt.figure(figsize=(10, 10))
for images, labels in train.take(1):
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))
        plt.title(class_names[labels[i]])
        plt.axis("off")
```



Model Development

I define a convolutional neural network (CNN) model using the keras library. The model consists of a sequence of layers that process and transform the input data to extract relevant features and make predictions.

The first layer in the model is a '*Rescaling layer*', which scales the input data by dividing each pixel value by 255. This helps to normalize the data and make it easier for the model to learn.

The model includes three '*Conv2D layers*', which apply a series of filters to the input data to extract features. The kernel_size argument specifies the size of the filters, and the padding argument specifies how to handle the edges of the input data. The activation argument specifies the activation function to use for the layer, and the kernel_initializer argument specifies the method used to initialize the weights of the filters.

The model also includes three '*MaxPooling2D layers*', which reduce the spatial dimensions of the data by applying a max pooling operation. This helps to reduce the complexity of the model and improve its ability to generalize to unseen data.

The model includes two '*Dropout layers*', which randomly set a portion of the input units to zero during training to prevent overfitting.

After the convolutional and max pooling layers, the model flattens the output data and passes it through a series of '*fully connected (dense) layers*', which are used to make the final classification decision. The model ends with a final dense layer with 4 units and a softmax activation function, which produces a probability distribution over the 4 classes.

The '*compile method*' is used to specify the loss function, optimizer, and metrics to use when training the model.

The '*summary method*' is used to display a summary of the model architecture and the number of parameters in each layer.

In [23]:
```
model = keras.models.Sequential()
model.add(keras.layers.experimental.preprocessing.Rescaling(1./255, input_shape=(128,128,3)))
model.add(keras.layers.Conv2D(filters=16,kernel_size=(3,3),padding='same',activation='relu',kernel_initializer=
model.add(keras.layers.MaxPooling2D(pool_size=(2,2)))


model.add(keras.layers.Conv2D(filters=32,kernel_size=(3,3),padding='same',activation='relu',kernel_initializer=
model.add(keras.layers.MaxPooling2D(pool_size=(2,2)))

model.add(keras.layers.Dropout(0.20))

model.add(keras.layers.Conv2D(filters=64,kernel_size=(3,3),padding='same',activation='relu',kernel_initializer=
model.add(keras.layers.MaxPooling2D(pool_size=(2,2)))

model.add(keras.layers.Dropout(0.25))
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(128,activation="relu",kernel_initializer="he_normal"))
model.add(keras.layers.Dense(64,"relu"))
model.add(keras.layers.Dense(4,"softmax"))
```

```
model.compile(loss="sparse_categorical_crossentropy", optimizer = "Adam",metrics=["accuracy"])
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 rescaling_1 (Rescaling)     (None, 128, 128, 3)       0

 conv2d_3 (Conv2D)           (None, 128, 128, 16)      448

 max_pooling2d_3 (MaxPooling  (None, 64, 64, 16)       0
 2D)

 conv2d_4 (Conv2D)           (None, 64, 64, 32)        4640

 max_pooling2d_4 (MaxPooling  (None, 32, 32, 32)       0
 2D)

 dropout_2 (Dropout)         (None, 32, 32, 32)        0

 conv2d_5 (Conv2D)           (None, 32, 32, 64)        18496

 max_pooling2d_5 (MaxPooling  (None, 16, 16, 64)       0
 2D)

 dropout_3 (Dropout)         (None, 16, 16, 64)        0

 flatten_1 (Flatten)         (None, 16384)             0

 dense_3 (Dense)             (None, 128)               2097280

 dense_4 (Dense)             (None, 64)                8256

 dense_5 (Dense)             (None, 4)                 260

=================================================================
Total params: 2,129,380
Trainable params: 2,129,380
Non-trainable params: 0
_________________________________________________________________
```

This method takes as input the training dataset and optional arguments such as the validation dataset, the number of epochs to train for, and the batch size.

In [24]:
```
hist = model.fit(train,validation_data=val,epochs=100, batch_size=64, verbose=1)
```

```
Epoch 1/100
80/80 [==============================] - 5s 49ms/step - loss: 1.1677 - accuracy: 0.4825 - val_loss: 1.0186 - va
l_accuracy: 0.5462
Epoch 2/100
80/80 [==============================] - 5s 62ms/step - loss: 0.9280 - accuracy: 0.5460 - val_loss: 0.8550 - va
l_accuracy: 0.6197
Epoch 3/100
80/80 [==============================] - 4s 40ms/step - loss: 0.8259 - accuracy: 0.6202 - val_loss: 0.7227 - va
l_accuracy: 0.6948
Epoch 4/100
80/80 [==============================] - 3s 36ms/step - loss: 0.7175 - accuracy: 0.6861 - val_loss: 0.7527 - va
l_accuracy: 0.6682
Epoch 5/100
80/80 [==============================] - 3s 38ms/step - loss: 0.6071 - accuracy: 0.7414 - val_loss: 0.5232 - va
l_accuracy: 0.8044
Epoch 6/100
80/80 [==============================] - 4s 52ms/step - loss: 0.4944 - accuracy: 0.7914 - val_loss: 0.3970 - va
l_accuracy: 0.8435
Epoch 7/100
80/80 [==============================] - 3s 36ms/step - loss: 0.3844 - accuracy: 0.8478 - val_loss: 0.4297 - va
l_accuracy: 0.8122
Epoch 8/100
80/80 [==============================] - 3s 36ms/step - loss: 0.3490 - accuracy: 0.8595 - val_loss: 0.2728 - va
l_accuracy: 0.9045
Epoch 9/100
80/80 [==============================] - 3s 37ms/step - loss: 0.2758 - accuracy: 0.8867 - val_loss: 0.1966 - va
l_accuracy: 0.9484
Epoch 10/100
80/80 [==============================] - 3s 39ms/step - loss: 0.2081 - accuracy: 0.9176 - val_loss: 0.2639 - va
l_accuracy: 0.8905
Epoch 11/100
80/80 [==============================] - 3s 40ms/step - loss: 0.1757 - accuracy: 0.9367 - val_loss: 0.1152 - va
l_accuracy: 0.9750
Epoch 12/100
80/80 [==============================] - 4s 48ms/step - loss: 0.1465 - accuracy: 0.9443 - val_loss: 0.1508 - va
l_accuracy: 0.9484
Epoch 13/100
80/80 [==============================] - 4s 49ms/step - loss: 0.1191 - accuracy: 0.9555 - val_loss: 0.0830 - va
l_accuracy: 0.9687
Epoch 14/100
```

```
80/80 [==============================] - 3s 37ms/step - loss: 0.1254 - accuracy: 0.9539 - val_loss: 0.0767 - va
l_accuracy: 0.9812
Epoch 15/100
80/80 [==============================] - 3s 38ms/step - loss: 0.0775 - accuracy: 0.9740 - val_loss: 0.0621 - va
l_accuracy: 0.9812
Epoch 16/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0814 - accuracy: 0.9719 - val_loss: 0.0618 - va
l_accuracy: 0.9844
Epoch 17/100
80/80 [==============================] - 3s 39ms/step - loss: 0.0818 - accuracy: 0.9727 - val_loss: 0.0542 - va
l_accuracy: 0.9781
Epoch 18/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0561 - accuracy: 0.9805 - val_loss: 0.0430 - va
l_accuracy: 0.9890
Epoch 19/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0616 - accuracy: 0.9768 - val_loss: 0.0360 - va
l_accuracy: 0.9906
Epoch 20/100
80/80 [==============================] - 3s 38ms/step - loss: 0.0573 - accuracy: 0.9789 - val_loss: 0.0554 - va
l_accuracy: 0.9797
Epoch 21/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0560 - accuracy: 0.9791 - val_loss: 0.0369 - va
l_accuracy: 0.9875
Epoch 22/100
80/80 [==============================] - 4s 46ms/step - loss: 0.0526 - accuracy: 0.9816 - val_loss: 0.0300 - va
l_accuracy: 0.9922
Epoch 23/100
80/80 [==============================] - 4s 43ms/step - loss: 0.0433 - accuracy: 0.9852 - val_loss: 0.0396 - va
l_accuracy: 0.9890
Epoch 24/100
80/80 [==============================] - 3s 38ms/step - loss: 0.0437 - accuracy: 0.9838 - val_loss: 0.0396 - va
l_accuracy: 0.9844
Epoch 25/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0376 - accuracy: 0.9871 - val_loss: 0.0346 - va
l_accuracy: 0.9890
Epoch 26/100
80/80 [==============================] - 3s 38ms/step - loss: 0.0321 - accuracy: 0.9883 - val_loss: 0.0323 - va
l_accuracy: 0.9906
Epoch 27/100
80/80 [==============================] - 3s 39ms/step - loss: 0.0543 - accuracy: 0.9812 - val_loss: 0.0477 - va
l_accuracy: 0.9859
Epoch 28/100
80/80 [==============================] - 3s 38ms/step - loss: 0.0344 - accuracy: 0.9893 - val_loss: 0.0377 - va
l_accuracy: 0.9875
Epoch 29/100
80/80 [==============================] - 3s 38ms/step - loss: 0.0265 - accuracy: 0.9912 - val_loss: 0.0412 - va
l_accuracy: 0.9859
Epoch 30/100
80/80 [==============================] - 5s 57ms/step - loss: 0.0228 - accuracy: 0.9920 - val_loss: 0.0285 - va
l_accuracy: 0.9906
Epoch 31/100
80/80 [==============================] - 4s 41ms/step - loss: 0.0343 - accuracy: 0.9883 - val_loss: 0.0318 - va
l_accuracy: 0.9922
Epoch 32/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0275 - accuracy: 0.9914 - val_loss: 0.0246 - va
l_accuracy: 0.9953
Epoch 33/100
80/80 [==============================] - 3s 41ms/step - loss: 0.0208 - accuracy: 0.9930 - val_loss: 0.0093 - va
l_accuracy: 0.9969
Epoch 34/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0192 - accuracy: 0.9936 - val_loss: 0.0162 - va
l_accuracy: 0.9906
Epoch 35/100
80/80 [==============================] - 3s 39ms/step - loss: 0.0424 - accuracy: 0.9840 - val_loss: 0.0613 - va
l_accuracy: 0.9844
Epoch 36/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0260 - accuracy: 0.9906 - val_loss: 0.0206 - va
l_accuracy: 0.9937
Epoch 37/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0267 - accuracy: 0.9910 - val_loss: 0.0245 - va
l_accuracy: 0.9922
Epoch 38/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0180 - accuracy: 0.9943 - val_loss: 0.0236 - va
l_accuracy: 0.9890
Epoch 39/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0169 - accuracy: 0.9939 - val_loss: 0.0276 - va
l_accuracy: 0.9922
Epoch 40/100
80/80 [==============================] - 4s 51ms/step - loss: 0.0186 - accuracy: 0.9941 - val_loss: 0.0151 - va
l_accuracy: 0.9953
Epoch 41/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0180 - accuracy: 0.9949 - val_loss: 0.0160 - va
l_accuracy: 0.9937
Epoch 42/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0234 - accuracy: 0.9914 - val_loss: 0.0265 - va
l_accuracy: 0.9906
Epoch 43/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0225 - accuracy: 0.9926 - val_loss: 0.0704 - va
l_accuracy: 0.9781
```

```
Epoch 44/100
80/80 [==============================] - 3s 38ms/step - loss: 0.0227 - accuracy: 0.9922 - val_loss: 0.0545 - va
l_accuracy: 0.9828
Epoch 45/100
80/80 [==============================] - 3s 41ms/step - loss: 0.0185 - accuracy: 0.9934 - val_loss: 0.0545 - va
l_accuracy: 0.9859
Epoch 46/100
80/80 [==============================] - 3s 38ms/step - loss: 0.0206 - accuracy: 0.9928 - val_loss: 0.0191 - va
l_accuracy: 0.9922
Epoch 47/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0349 - accuracy: 0.9873 - val_loss: 0.0501 - va
l_accuracy: 0.9844
Epoch 48/100
80/80 [==============================] - 3s 38ms/step - loss: 0.0280 - accuracy: 0.9910 - val_loss: 0.0335 - va
l_accuracy: 0.9875
Epoch 49/100
80/80 [==============================] - 4s 51ms/step - loss: 0.0120 - accuracy: 0.9965 - val_loss: 0.0615 - va
l_accuracy: 0.9890
Epoch 50/100
80/80 [==============================] - 3s 41ms/step - loss: 0.0104 - accuracy: 0.9965 - val_loss: 0.0123 - va
l_accuracy: 0.9937
Epoch 51/100
80/80 [==============================] - 4s 44ms/step - loss: 0.0277 - accuracy: 0.9906 - val_loss: 0.0199 - va
l_accuracy: 0.9937
Epoch 52/100
80/80 [==============================] - 4s 44ms/step - loss: 0.0187 - accuracy: 0.9924 - val_loss: 0.0342 - va
l_accuracy: 0.9906
Epoch 53/100
80/80 [==============================] - 3s 40ms/step - loss: 0.0158 - accuracy: 0.9949 - val_loss: 0.0252 - va
l_accuracy: 0.9922
Epoch 54/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0114 - accuracy: 0.9951 - val_loss: 0.0254 - va
l_accuracy: 0.9890
Epoch 55/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0233 - accuracy: 0.9920 - val_loss: 0.0294 - va
l_accuracy: 0.9906
Epoch 56/100
80/80 [==============================] - 4s 46ms/step - loss: 0.0197 - accuracy: 0.9928 - val_loss: 0.0237 - va
l_accuracy: 0.9906
Epoch 57/100
80/80 [==============================] - 4s 42ms/step - loss: 0.0125 - accuracy: 0.9951 - val_loss: 0.0406 - va
l_accuracy: 0.9859
Epoch 58/100
80/80 [==============================] - 4s 43ms/step - loss: 0.0150 - accuracy: 0.9955 - val_loss: 0.0145 - va
l_accuracy: 0.9969
Epoch 59/100
80/80 [==============================] - 3s 38ms/step - loss: 0.0181 - accuracy: 0.9937 - val_loss: 0.0164 - va
l_accuracy: 0.9953
Epoch 60/100
80/80 [==============================] - 3s 40ms/step - loss: 0.0104 - accuracy: 0.9971 - val_loss: 0.0492 - va
l_accuracy: 0.9859
Epoch 61/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0155 - accuracy: 0.9949 - val_loss: 0.0182 - va
l_accuracy: 0.9922
Epoch 62/100
80/80 [==============================] - 4s 41ms/step - loss: 0.0101 - accuracy: 0.9965 - val_loss: 0.0246 - va
l_accuracy: 0.9906
Epoch 63/100
80/80 [==============================] - 3s 40ms/step - loss: 0.0134 - accuracy: 0.9951 - val_loss: 0.0536 - va
l_accuracy: 0.9875
Epoch 64/100
80/80 [==============================] - 4s 45ms/step - loss: 0.0060 - accuracy: 0.9980 - val_loss: 0.1002 - va
l_accuracy: 0.9765
Epoch 65/100
80/80 [==============================] - 3s 38ms/step - loss: 0.0213 - accuracy: 0.9936 - val_loss: 0.0361 - va
l_accuracy: 0.9890
Epoch 66/100
80/80 [==============================] - 4s 47ms/step - loss: 0.0186 - accuracy: 0.9932 - val_loss: 0.0198 - va
l_accuracy: 0.9937
Epoch 67/100
80/80 [==============================] - 4s 42ms/step - loss: 0.0095 - accuracy: 0.9969 - val_loss: 0.0115 - va
l_accuracy: 0.9937
Epoch 68/100
80/80 [==============================] - 4s 42ms/step - loss: 0.0198 - accuracy: 0.9936 - val_loss: 0.0299 - va
l_accuracy: 0.9906
Epoch 69/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0231 - accuracy: 0.9928 - val_loss: 0.0427 - va
l_accuracy: 0.9890
Epoch 70/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0126 - accuracy: 0.9957 - val_loss: 0.0286 - va
l_accuracy: 0.9922
Epoch 71/100
80/80 [==============================] - 4s 42ms/step - loss: 0.0097 - accuracy: 0.9965 - val_loss: 0.0588 - va
l_accuracy: 0.9781
Epoch 72/100
80/80 [==============================] - 4s 51ms/step - loss: 0.0121 - accuracy: 0.9969 - val_loss: 0.0245 - va
l_accuracy: 0.9906
Epoch 73/100
80/80 [==============================] - 3s 39ms/step - loss: 0.0226 - accuracy: 0.9932 - val_loss: 0.0322 - va
```

```
l_accuracy: 0.9906
Epoch 74/100
80/80 [==============================] - 3s 36ms/step - loss: 0.0133 - accuracy: 0.9963 - val_loss: 0.0257 - va
l_accuracy: 0.9953
Epoch 75/100
80/80 [==============================] - 4s 43ms/step - loss: 0.0093 - accuracy: 0.9973 - val_loss: 0.0156 - va
l_accuracy: 0.9937
Epoch 76/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0079 - accuracy: 0.9969 - val_loss: 0.0100 - va
l_accuracy: 0.9969
Epoch 77/100
80/80 [==============================] - 4s 43ms/step - loss: 0.0123 - accuracy: 0.9959 - val_loss: 0.0257 - va
l_accuracy: 0.9890
Epoch 78/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0112 - accuracy: 0.9957 - val_loss: 0.0046 - va
l_accuracy: 1.0000
Epoch 79/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0128 - accuracy: 0.9961 - val_loss: 0.0067 - va
l_accuracy: 0.9969
Epoch 80/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0043 - accuracy: 0.9986 - val_loss: 0.0366 - va
l_accuracy: 0.9875
Epoch 81/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0072 - accuracy: 0.9979 - val_loss: 0.0209 - va
l_accuracy: 0.9922
Epoch 82/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0137 - accuracy: 0.9949 - val_loss: 0.0293 - va
l_accuracy: 0.9890
Epoch 83/100
80/80 [==============================] - 3s 39ms/step - loss: 0.0127 - accuracy: 0.9955 - val_loss: 0.0301 - va
l_accuracy: 0.9906
Epoch 84/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0138 - accuracy: 0.9955 - val_loss: 0.0158 - va
l_accuracy: 0.9937
Epoch 85/100
80/80 [==============================] - 3s 38ms/step - loss: 0.0207 - accuracy: 0.9924 - val_loss: 0.0432 - va
l_accuracy: 0.9890
Epoch 86/100
80/80 [==============================] - 3s 38ms/step - loss: 0.0165 - accuracy: 0.9939 - val_loss: 0.0276 - va
l_accuracy: 0.9922
Epoch 87/100
80/80 [==============================] - 3s 36ms/step - loss: 0.0205 - accuracy: 0.9934 - val_loss: 0.0206 - va
l_accuracy: 0.9922
Epoch 88/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0177 - accuracy: 0.9941 - val_loss: 0.0165 - va
l_accuracy: 0.9906
Epoch 89/100
80/80 [==============================] - 3s 38ms/step - loss: 0.0239 - accuracy: 0.9920 - val_loss: 0.0445 - va
l_accuracy: 0.9859
Epoch 90/100
80/80 [==============================] - 3s 38ms/step - loss: 0.0224 - accuracy: 0.9912 - val_loss: 0.0208 - va
l_accuracy: 0.9922
Epoch 91/100
80/80 [==============================] - 3s 39ms/step - loss: 0.0158 - accuracy: 0.9947 - val_loss: 0.0285 - va
l_accuracy: 0.9890
Epoch 92/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0124 - accuracy: 0.9949 - val_loss: 0.0088 - va
l_accuracy: 0.9953
Epoch 93/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0099 - accuracy: 0.9959 - val_loss: 0.0059 - va
l_accuracy: 0.9969
Epoch 94/100
80/80 [==============================] - 3s 38ms/step - loss: 0.0079 - accuracy: 0.9977 - val_loss: 0.0370 - va
l_accuracy: 0.9906
Epoch 95/100
80/80 [==============================] - 3s 39ms/step - loss: 0.0064 - accuracy: 0.9977 - val_loss: 0.0104 - va
l_accuracy: 0.9937
Epoch 96/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0087 - accuracy: 0.9973 - val_loss: 0.0133 - va
l_accuracy: 0.9953
Epoch 97/100
80/80 [==============================] - 3s 39ms/step - loss: 0.0084 - accuracy: 0.9971 - val_loss: 0.0393 - va
l_accuracy: 0.9922
Epoch 98/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0058 - accuracy: 0.9982 - val_loss: 0.0126 - va
l_accuracy: 0.9953
Epoch 99/100
80/80 [==============================] - 3s 37ms/step - loss: 0.0031 - accuracy: 0.9990 - val_loss: 0.0318 - va
l_accuracy: 0.9937
Epoch 100/100
80/80 [==============================] - 3s 39ms/step - loss: 0.0133 - accuracy: 0.9967 - val_loss: 0.0220 - va
l_accuracy: 0.9937
```

The fit method above returns a History object that contains information about the training process, including the values of the loss and metrics at each epoch. The hist variable in this code will contain this History object.

I used the history attribute of the History object to access the training and validation loss and metric values.

In [25]:
```
get_ac = hist.history['accuracy']
get_los = hist.history['loss']
val_acc = hist.history['val_accuracy']
val_loss = hist.history['val_loss']
epochs = range(len(get_ac))
```

Used the history attribute to plot accuracy and loss for training data over time:

In [26]:
```
plt.plot(epochs, get_ac, 'g', label='Accuracy of Training data')
plt.plot(epochs, get_los, 'r', label='Loss of Training data')
plt.title('Training data accuracy and loss')
plt.legend(loc=0)
plt.figure()
```

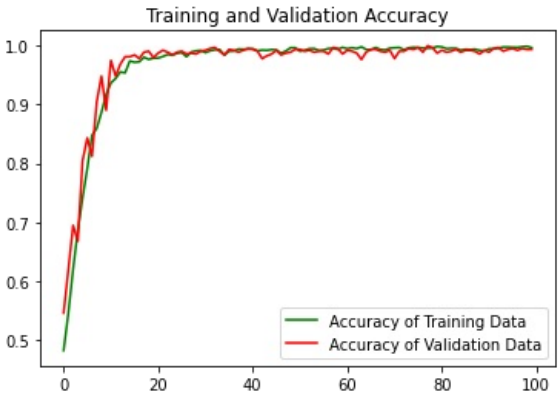
Out[26]:
```
<Figure size 432x288 with 0 Axes>
```


```
<Figure size 432x288 with 0 Axes>
```

Used the history attribute to plot training and validation accuracy over time:

In [27]:
```
plt.plot(epochs, get_ac, 'g', label='Accuracy of Training Data')
plt.plot(epochs, val_acc, 'r', label='Accuracy of Validation Data')
plt.title('Training and Validation Accuracy')
plt.legend(loc=0)
plt.figure()
```

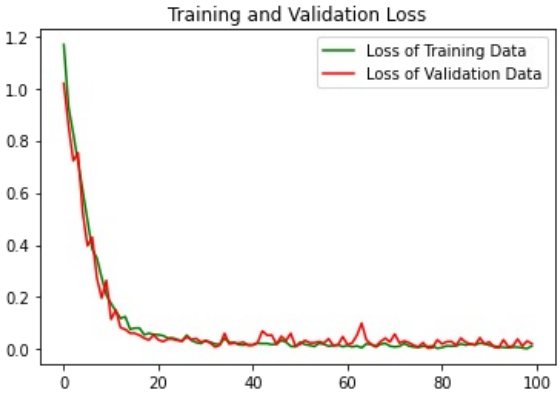
Out[27]:
```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```

Used the history attribute to plot training and validation loss over time:

In [28]:
```
plt.plot(epochs, get_los, 'g', label='Loss of Training Data')
plt.plot(epochs, val_loss, 'r', label='Loss of Validation Data')
plt.title('Training and Validation Loss')
plt.legend(loc=0)
plt.figure()
plt.show()
```




```
<Figure size 432x288 with 0 Axes>
```

I used the evaluation method to evaluate the generalization performance of the model on invisible data and apply it to the model on loss and accuracy and obtain a prediction result. The obtained value gives me information about how successful the model I developed is on the test data.

In [29]:
```
loss, accuracy = model.evaluate(test)
```

```
11/11 [==============================] - 0s 14ms/step - loss: 0.0451 - accuracy: 0.9891
```

Result of Model

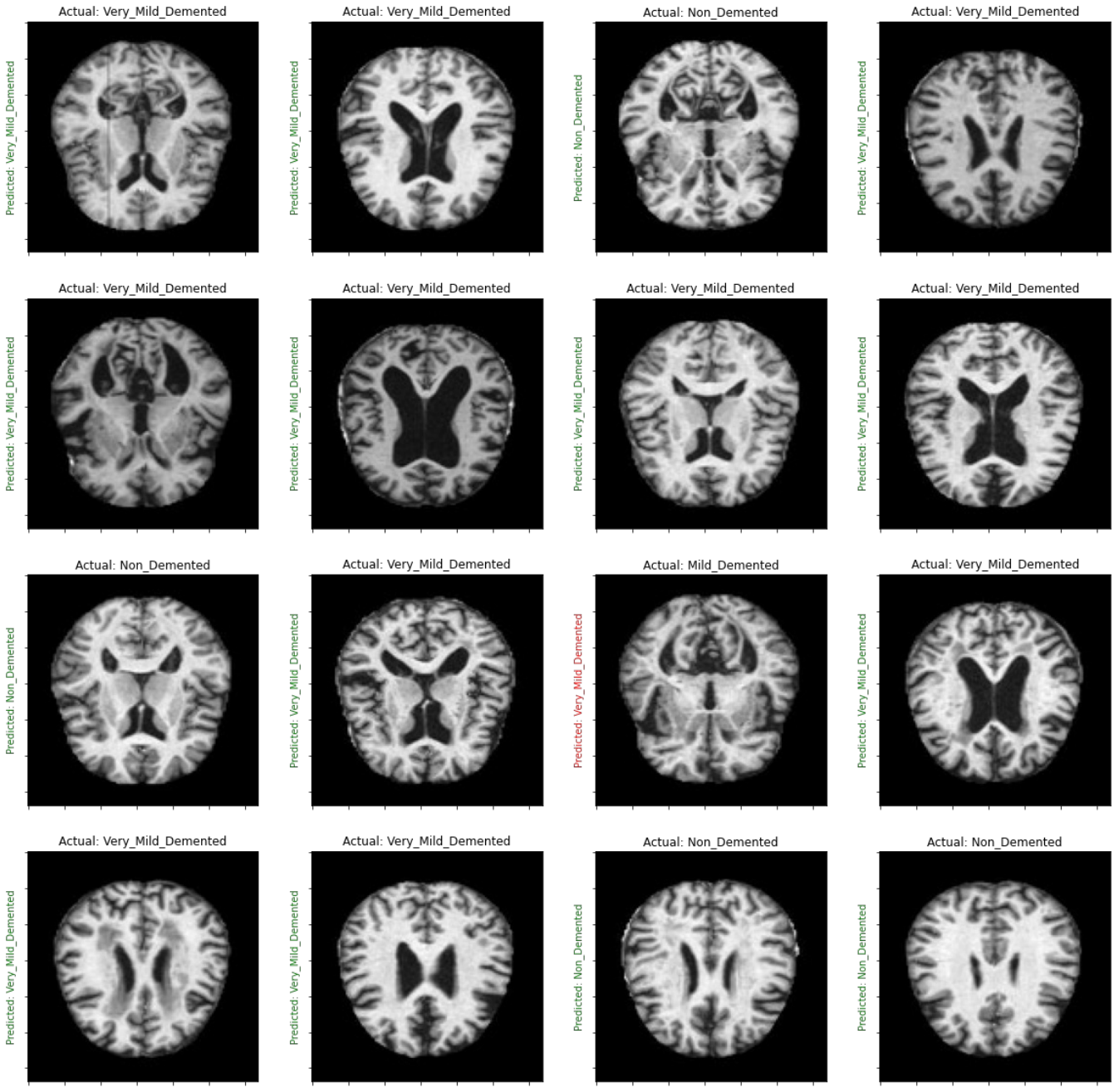
This code displays a grid of 16 images from the test dataset, with the actual and predicted class names displayed for each image. The predicted class names displayed in green if the prediction was correct, or in red if the prediction was incorrect. This is a useful way to visualize the performance of the model on the test set and identify any misclassified images.

As can be seen in my model result below, the label (class) of only 1 picture out of 16 was assigned incorrectly, so the label turned red.

In [30]:
```
plt.figure(figsize=(20, 20))
for images, labels in test.take(1):
    for i in range(16):
        ax = plt.subplot(4, 4, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))
        predictions = model.predict(tf.expand_dims(images[i], 0))
        score = tf.nn.softmax(predictions[0])
        if(class_names[labels[i]]==class_names[np.argmax(score)]):
            plt.title("Actual: "+class_names[labels[i]])
            plt.ylabel("Predicted: "+class_names[np.argmax(score)],fontdict={'color':'green'})

        else:
            plt.title("Actual: "+class_names[labels[i]])
            plt.ylabel("Predicted: "+class_names[np.argmax(score)],fontdict={'color':'red'})
        plt.gca().axes.yaxis.set_ticklabels([])
        plt.gca().axes.xaxis.set_ticklabels([])
```

```
1/1 [==============================] - 0s 75ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 15ms/step
```



Methodology

Convolutional neural networks (CNNs) are particularly well-suited for processing data with a grid-like topology that refers to a structure in which data is arranged in a grid or matrix-like format because they are designed to learn features from raw data using a process called convolution.

In a CNN, the filters used in the convolutional layers are typically smaller than the input data and are moved across the input data using a sliding window approach. This allows the CNN to learn features at different scales and locations in the input data.

1. Preprocess the data: I tried to preprocess the input data to make it more suitable for a CNN model. It involves scaling the data, normalizing the data, or applying other transformations to the data.
2. Split the data into training and test sets: I split the available data into a training set, which is used to train the model, and a test set,

which is used to evaluate the model's performance. Then a validation set which provides an unbiased evaluation of a model fit on the training data set while tuning the model's hyperparameters.

3. Define the model architecture:I chose the type and number of layers to use in the model, as well as the specific hyperparameters for each layer (e.g., number of filters, kernel size, etc.).
4. Compile the model: After the model architecture defined and the data has been preprocessed, the model is compiled using the compile method. This involves specifying the loss function, optimizer, and metrics to use during training.
5. Train the model: I trained the model using the fit method, which takes the training data as input. During training, the model's parameters are adjusted to minimize the loss function. I controlled the training process using various hyperparameters, such as the batch size and the number of epochs.
6. Evaluate the model: After training, the model is typically evaluated on a separate test set to gauge its performance. This is done using the evaluate method, which returns the loss and accuracy metrics for the test set.

Thanks to the items I used in this study, I learned how I can achieve good results with different models by designing a model for a real health problem.

Future Work

I will tried to improve another model on this dataset. For example, using the U-net algorithm, I can get it to give more reliable accuracy with different machine learning optimizations like Adam, Adagrad, RMSprop.

It is very important to detect Alzheimer's disease at an early stage. When it correctly guesses the MRI image, which is in the very mild demented class, it can give a warning to the system.