Predicting Alzheimer's Disease Using Deep Learning Artificial Intelligence Together with a Pre-Trained VGG19 and Inception_v3 Models

Paul Gabriel TEODORESCU[1], Silvia OVREIU[1,2], Mădălina ZAMFIR[1], Cristian ȚÎRLEA[1]
[1]National Institute for Research & Development in Informatics - ICI Bucharest, Romania
[2]University Polithenica of Bucharest, Romania
paul.teodorescu@ici.ro, silvia.ovreiu@ici.ro, madalina.zamfir@ici.ro, cristian.tirlea@ici.ro

This paper presents two experiments in which, using artificial intelligence (specifically Deep Learning with convolutional neural networks), we were able to predict Alzheimer's disease based on MRI images. In order to have better results and to minimize the computational effort in the laboratory, two pre-trained AI models were used, models trained previously on more than a million images from the ImageNet database (which provide tens of millions of clean, labelled and sorted images). The top-layers of the models were trained, for our specific task of Alzheimer's prediction, with 500 public MRI images from Kaggle, an online community of data scientists and machine learning engineers and a subsidiary of Google. In this paper we describe the code used in the laboratory for the specific task.
Keywords: *MRI images, ImageNet, Feature extractor, Demented, Accuracy, Pre-trained, Target, Convolutional, Matrix*
DOI: 10.24818/issn14531305/28.2.2024.02

1 Introduction

Predicting Alzheimer's based on MRI (Magnetic Resonance Imaging) images is a promising area of medical research with the potential to revolutionize the diagnosis and treatment of this debilitating condition. MRI images provide detailed insight into the structure and functioning of the brain, allowing researchers to identify early signs of Alzheimer's disease with increased accuracy and sensitivity. Using machine learning and artificial intelligence algorithms, researchers can analyze the volume, shape and connectivity of different brain regions in MRI images to detect changes associated with Alzheimer's disease. This approach can identify subtle abnormalities in brain structure that may be early indicators of cognitive decline associated with the disease.

However, it is important to emphasize that the prediction of Alzheimer's based on MRI images is still in the research and development stages [1]. More studies and validation are needed to confirm the accuracy and reliability of these diagnostic techniques. Ethical and privacy issues also need to be considered regarding the use of medical data and artificial intelligence algorithms in medical practice. Despite the challenges and ethical questions, advances in MRI imaging and artificial intelligence promise to offer significant opportunities for the early diagnosis and management of Alzheimer's disease [2], which could lead to significant improvements in the quality of life of patients and their families.

In this paper, two experiments will be done for Alzheimer predictions using AI with public data. In what is presented, it is clearly revealed that AI is in an active phase of research and development and that this field will revolutionize the way these conditions are diagnosed, treated and managed [3].

In our experiments, for Alzheimer's prediction we will use MRI images. Work is done in the Google Colab environment, which offers many advantages for developing and running Python code, especially in the areas of machine learning, deep learning and data analysis. Google Colab also offers free access to powerful computing resources (including GPU graphics processors, TPU tensor processors that can significantly speed up the training time of machine learning models. Since these experiments were team-based, Google Colab (via Google Drive) enabled collaboration on code development and data access. Furthermore, Google Colab comes pre-loaded

with a number of popular Python libraries and packages, such as TensorFlow, Keras, NumPy and Pandas, facilitating rapid development and experimentation and thus
eliminating the need to install and configure software on local devices.
In the first experiment, we'll use the pre-trained VGG19 model, which is a convolutional neural network (CNN) model developed by the Visual Geometry Group (VGG) at the University of Oxford. This architecture has been trained on a massive set of images from the ImageNet database. VGG19 is known for its simplicity and depth, it is often used as a pre-trained model, has 19 layers, including 16 convolutional layers and 3 fully connected layers. The depth of the network is one of its distinguishing features and is known for its ability to capture complex details from images. The Google Colab environment in which we worked, helped us cope with the high computational demands required by this architecture. What is important to understand in these experiments is that instead of training a neural network built by us from scratch, a pre-trained model is used. This model has weights updated previously with large datasets from ImageNet (ImageNet data). So, in our experiment we do not want to update the weights of the lower layers during the training operation and therefore they will be frozen. The top layers are to be eliminated and replaced with custom layers. Some pieces of code later on will make this clarification:

```
include_top=False together with layer.trainable = False.
```

It does not make sense to update the base VGG19 model weights because they have already been updated with large datasets. The lower layers of the network have learned from these large datasets and only the upper layers (i.e. the fully connected ones) are trained in the present experiment, on a public dataset that will be located on Google Drive. The network used will therefore act as a *fixed feature extractor* and this process is called "*transfer learning*" [2]: what the VGG19 [5] network has learned (being trained on a huge ImageNet dataset), it will be transferred to the new task which is the Alzheimer's prediction.
The following explains how the code was written. This code will classify the MRI images stored in the *Alzheimer's-Disease* folder on Google Drive. To access this data (which are *.jpg* images), the Google Drive is mounted within the workspace of the current Colab notebook session:

```
from google.colab import drive
drive.mount('/content/drive')
```

We'll use TensorFlow which has become one of the most popular and influential machine learning platforms having artificial intelligence libraries available. The TensorFlow module contains implementations for Keras, a high-level library for building and training neural networks.

2 Components of TensorFlow's Keras API

In order to build neural network, is necessary to import various components. First, we'll start to import specific *layer* classes from TensorFlow and Keras module. *Layer* classes in deep learning frameworks (like TensorFlow and Keras) are fundamental building-blocks used to construct neural networks. Each *layer* class represents a specific type of mathematical operation or transformation that can be applied to the input data to produce output data. These layers are organized sequentially to form the architecture of the neural network. *Layer* classes abstract away the complexity of implementing neural network components such as neurons, activation functions and weight matrices. Instead of manually coding these components, we can use pre-implemented *layer* classes, which simplifies the process of building neural networks:

```
from tensorflow.keras.layers import Input, Lambda, Dense, Flatten, AveragePooling2D,
Dropout
```

Input layer is used to instantiate a Keras tensor [3] , *Lambda* layer is used to implement custom operations or functions, *Dense* layer is used as a fully connected layer for output in classification tasks, *Flatten* layer is used to flatten the input tensor into a one dimensional array, a layer to perform 2D average pooling operation over the input data, an *AveragePooling2* layer and the *Dropout* layer to prevent overfitting through regularization technique (randomly sets a fraction of input units to zero during training).
Having now the high-level interface for constructing our architecture of the neural network, we need to import several modules and functions from TensorFlow and Keras, which are useful for working with deep learning models, especially convolutional neural networks (CNNs). One of the fundamental building-block for defining and organizing neural network architectures is the Model class. The Model class allows us to compose neural network architectures by specifying the input and output layers. It provides a high-level interface for defining and managing this hierarchical structure, making it easy to conceptualize and implement complex architectures. Also, the Model class provides built-in methods for training and evaluating the model on training, validation and test data. It abstracts away the details of the training loop, including forward and backward passes, parameter updates and performance metrics calculation, simplifying the process of training and evaluating neural networks. Considering that we will address the *transfer learning* process in this paper, it is worth mentioning that Model class facilitates transfer learning and fine-tuning by providing methods for freezing and unfreezing specific layers or groups of layers, as we'll do in our experiments:

```
from tensorflow.keras.models import Model
```

Next, we'll import the function *load_model* from TensorFlow or Keras because we'll use and load a pre-trained model. Also, this function allows us not only to load pre-trained models, but also can serve as a starting point for transfer learning, saving time and computational resources. It must be said that *load_model* function [4] abstracts away the details of loading and initializing neural network models from saved files (the pre-trained model will be saved in a file format supported by the framework, typically in HDF5 format). The function handles the process of reconstructing the model architecture and loading the trained weights and other configuration parameters, making it easy to use saved models with minimal code:

```
from tensorflow.keras.models import load_model
```

As was mentioned earlier, we decided to use VGG19[5] in our experiments. Therefore, we need to import this pre-trained model (trained on the ImageNet dataset, a large dataset containing millions of labeled images across thousands of categories) provided by TensorFlow:

```
from tensorflow.keras.applications import VGG19
```

Before feeding our data into VGG19 model, it is necessary to apply some pre-processing operations which typically involve normalizing the pixel values of the input images to ensure they are in the correct range expected by the model. For doing this, the *preprocess_input* function is used specifically designed for the VGG19 model:

```
from tensorflow.keras.applications.vgg19 import preprocess_input
```

Considering that we are working with images, we need to import the *image* module from TensorFlow's Keras preprocessing utilities. The *image* module provides various functions and classes. Here are some common tasks that we can perform using the *image* module:

- Loading Images: The *load_img()* function can be used to load images from disk into Python variables.
- Resizing Images: The *img_to_array()* function converts images loaded using load_img() into NumPy arrays, and the array_to_img() function converts NumPy arrays back into images. You can also use the *resize ()* function to resize images to a specific size.
- Data Augmentation [7]: The *ImageDataGenerator* class provides methods for performing data augmentation, such as random rotations, shifts, flips, and zooms. Data augmentation is commonly used to increase the diversity of the training data and improve the robustness of deep learning models.
- Preprocessing for Models [8]: Pretrained deep learning models often require input images to be preprocessed in a specific way before making predictions. The *preprocess_input()* function applies preprocessing operations to input images to ensure that they are compatible with a particular model's requirements.

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

As it was mentioned before, only the fully connected layers at the end of the network are to be trained with our custom data consisting of 500 RMN images. These fully connected layers are responsible for learning high-level features from the raw pixel data provided as input. So, doing training only for the fully connected layers with custom data, we can adapt the model to our specific task or dataset while benefiting from the generalization power of the pre-trained convolutional layers. In other words, the learned representations of the convolutional base layers are preserved. It was imported the pre-trained VGG19 model discussed earlier and also the *image* which provides functions and utilities for working with images during pre-processing. The class *ImageDataGenerator* is used for generating batches of images during the training, including applying data augmentation to improve the model's generalization. Also, we'll import necessary libraries and functions for evaluating and visualizing the performance of machine learning models, particularly classification models. They allow for the computation of metrics such as confusion matrix, ROC (Receiver operating characteristic) curve and AUC-ROC score (Area under the ROC Curve), as well as for creating visualizations using Seaborn and Matplotlib [5]:

```
from sklearn.metrics import confusion_matrix, roc_curve, roc_auc_score
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt
```

The *confusion matrix* is particularly useful for assessing the performance of a classification algorithm when the true values of the target variable are known. It provides a more detailed understanding of how well the model is performing in terms of different types of errors it makes. The *numpy* is a fundamental package for scientific computing with Python, the Seaborn library (sns) is a data visualization library based on Matplotlib which make drawings and informative statistical graphics. Pyplot is a module within Matplotlib that provides a MATLAB-like plotting interface.

3 Images pre-processing, visualization and construction of the final model

Next, the data must be pre-processed so the path to the data is provided for this task. Also, we're using the *glob* function to search for all files within these directories. The * wildcard character is used to match any file name in the directories. The variable *Non-Demfiles* and *Demfiles* will contain lists of file paths corresponding to the files found in the directories specified by *nondement_path* and dement_path, respectively.

```
dement_path = '/gdrive/My Drive/Alzheimer_Project/train/MildDemented'
nondement_path = '/gdrive/My Drive/Alzheimer_Project/train/NonDemented'
```

```
NonDemfiles = glob( nondement_path +'/*' )
Demfiles = glob( dement_path + '/*' )
```

Iterating through these two sets of image files, Demfiles and NonDemfiles, we are performing the following steps:

- Reading the image using OpenCV's *cv2.imread()* function.
- Converting the colour space from BGR to RGB using *cv2.cvtColor().*
- Resizing the image to a fixed size of 229x229 pixels using *cv2.resize().*
- Appending the resized image to either *Dem_images* or *NonDem_images* based on whether the file belongs to the "Demented" or "NonDemented" category.
- Appending the corresponding label ("Demented" or "NonDemented") to either *Dem_labels* or *NonDem_labels*.

This process prepared the dataset in order to be categorized as either "Demented" or "NonDemented". After running the code below, we'll have two lists:

- *Dem_images*: Contains resized RGB images categorized as "Demented".
- *Dem_labels*: Contains corresponding labels ("Demented") for the images in *Dem_images*.

And similarly, we'll do with NonDem images. These lists will be used for training our model, with the images as input features and the labels as target values:

```
Dem_labels = []
NonDem_labels = []
Dem_images=[]
NonDem_images=[]
for i in range(len(Demfiles)):
  image = cv2.imread(Demfiles[i])
  image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
  image = cv2.resize(image,(229,229))
  Dem_images.append(image)
  Dem_labels.append('Demented')
for i in range(len(NonDemfiles)):
  image = cv2.imread(NonDemfiles[i])
  image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
  image = cv2.resize(image,(229,229))
  NonDem_images.append(image)
  NonDem_labels.append('NonDemented')
```

To have some visuals, we are using *plot_images* function to visualize a grid of images [6]. It takes a list of images (*images*) and a title (*title*) as input, and then plots these images in a grid layout.

```
def plot_images(images, title):
    nrows, ncols = 5, 8
    figsize = [10, 6]
    fig, ax = plt.subplots(nrows=nrows, ncols=ncols, figsize=figsize, facecolor=(1,
1, 1))
    for i, axi in enumerate(ax.flat):
    axi.imshow(images[i])
```

```
        axi.set_axis_off()

    plt.suptitle(title, fontsize=24)
    plt.tight_layout(pad=0.2, rect=[0, 0, 1, 0.9])
    plt.show()
plot_images(Dem_images, 'Demented Alzheimers Scan')
plot_images(NonDem_images, 'NonDemented Alzheimers Scan')
```

In figure 1 is presented a grid of images specific to *Demented* Alzheimers and *NonDemented* Alzheimers scan:

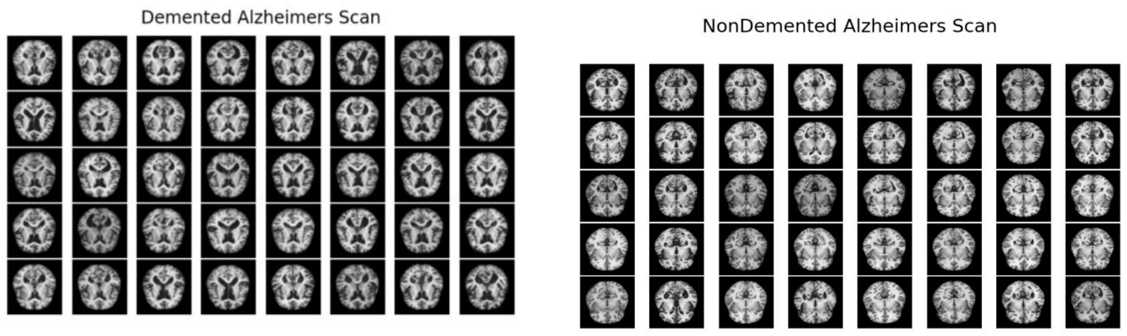


Fig. 1. A grid of images with the specified title

Normalizing pixel values to the range [0, 1] can help improve the training process and convergence of neural networks [7]. By dividing each pixel value by 255, we're scaling the values to be in the range [0, 1], as the original pixel values typically range from 0 to 255 (assuming 8-bit color depth).

```
Dem_images = np.array(Dem_images) / 255
NonDem_images = np.array(NonDem_images) / 255
```

We mentioned that the images we are working with (and which constitute the training dataset) are divided in two classes/categories: with and without dementia, as you can guess from the names of the folders that contain them. The features and the label of each photo are contained in variables *X_train* (for features) and *y_train* (for labels) which are vectors/arrays input data. The *X_train* represent the input features or attribute of the data (is a matrix of pixel values) and *y_train* contains the corresponding labels for the images in X_train, indicating their class or category.

Typically, when we have dataset with input features (X) and corresponding labels or target variables (y), we want to split it into separate sets for training your model and testing its performance. A commonly function used in machine learning to do this is *train_test_split*.
Dem_images represents the features (input data, images) associated with demented data and *Dem_labels* are the corresponding labels (output data, class labels) associated with the *demented* data:

```
Dem_x_train, Dem_x_test, Dem_y_train, Dem_y_test = train_test_split(
    Dem_images, Dem_labels, test_size=0.2)
```

And for *non_demented* data:

```
NonDem_x_train, NonDem_x_test, NonDem_y_train, NonDem_y_test = train_test_split(
    NonDem_images, NonDem_labels, test_size=0.2)
```

The parameter test_size=0.2 specifies that 20% of the data will be reserved for testing, while the remaining 80% will be used for training. After splitting the data, we have the

following variable:
Dem_x_train contains the features (images) of the demented data used for training, *Dem_x_test* contains the features (images) of the demented data used for testing, *Dem_y_train* contains the corresponding labels of the demented data used for training, *Dem_y_test* contains the corresponding labels of the demented data used for testing. Similarly, the non-demented data split process will take place. The resulting data can now be used for training and for evaluating machine learning models separately on each dataset. The data preparation (preprocessing step) continue with concatenation tasks resulting new variables:

```
X_train = np.concatenate((NonDem_x_train, Dem_x_train), axis=0)
X_test = np.concatenate((NonDem_x_test, Dem_x_test), axis=0)
y_train = np.concatenate((NonDem_y_train, Dem_y_train), axis=0)
y_test = np.concatenate((NonDem_y_test, Dem_y_test), axis=0)
```

The first line concatenates the features (input data, images) from both the dem (*Dem_x_train*) and non-dem (*NonDem_x_train*) training sets along the specified axis (axis=0 indicates concatenation along the rows). The resulting X_train contains the combined features for training. Similarly, we'll do with Training labels *(y_train*), with Testing data (resulting *X_test*) and with Testing labels (*y_train*). After that, is necessary to binarize the labels converting them into binary representation. Then, the function *to_categorical* from Keras is applied to convert the binarized labels into one-hot encoded vectors[8].

```
y_train = LabelBinarizer().fit_transform(y_train)
y_train = to_categorical(y_train)

y_test = LabelBinarizer().fit_transform(y_test)
y_test = to_categorical(y_test)
```

The plotting process shows the images from Table 2.

```
plot_images(Dem_x_train, 'X_train')
plot_images(Dem_x_test, 'X_test')
```

In figure 2 is presented a grid of images with the variable X_train and X_test:

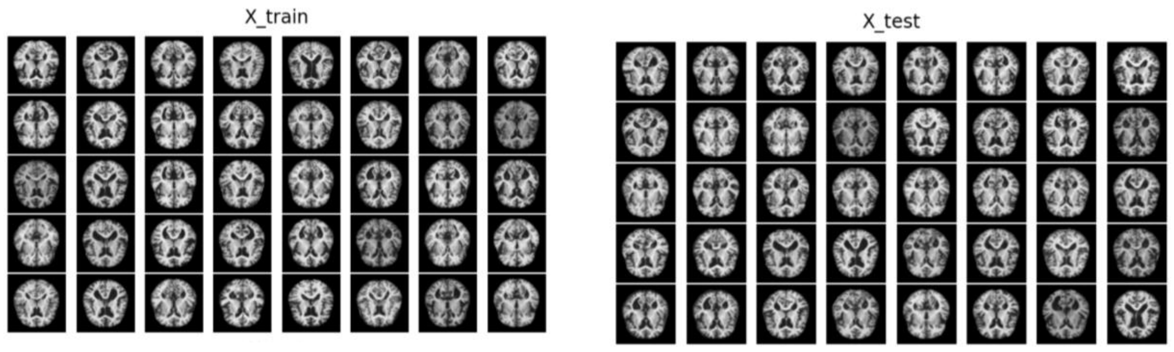


Fig. 2. A grid of images with the variable name as title

We are now ready to build a new neural network model for classification using transfer learning with the VGG19 architecture as a base.

```
VGGmodel = VGG19(weights="imagenet", include_top=False,
    input_tensor=Input(shape=(229, 229, 3)))
```

This line creates an instance of the VGG19 model using pre-trained weights from the ImageNet dataset (weights="imagenet"). It sets *include_top=False* to exclude the fully connected layers at the top of the VGG19 network, as these will be replaced with custom layers [9]. The *input_tensor* parameter defines the shape of the input data, which is expected to be an image with dimensions (229, 229, 3) (height, width, channels). We need now to define additional layers to be added on top of the VGG19 base model: *Flatten* layer is used to flatten the output tensor from the VGG19 base model, *Dropout* layer is applied with a dropout rate of 0.4 to prevent overfitting, *Dense* layer with 2 units and softmax activation function. This is added as the output layer for binary classification (2 classes):

```
outputs = VGGmodel.output
outputs = Flatten(name="flatten")(outputs)
outputs = Dropout(0.4)(outputs)
outputs = Dense(2, activation="softmax")(outputs)
```

It is time to construct the final model by specifying the inputs (defined by the input layer of the VGG19 model) and the outputs (defined by the custom layers):

```
model = Model(inputs=VGGmodel.input, outputs=outputs)
```

As was mentioned, only the custom layers added on top of the VGG19 model will be trained. So, we'll loop through all the layers of the VGG19 base model to set them to non-trainable, freezing their weights during training.

```
for layer in VGGmodel.layers:
    layer.trainable = False
```

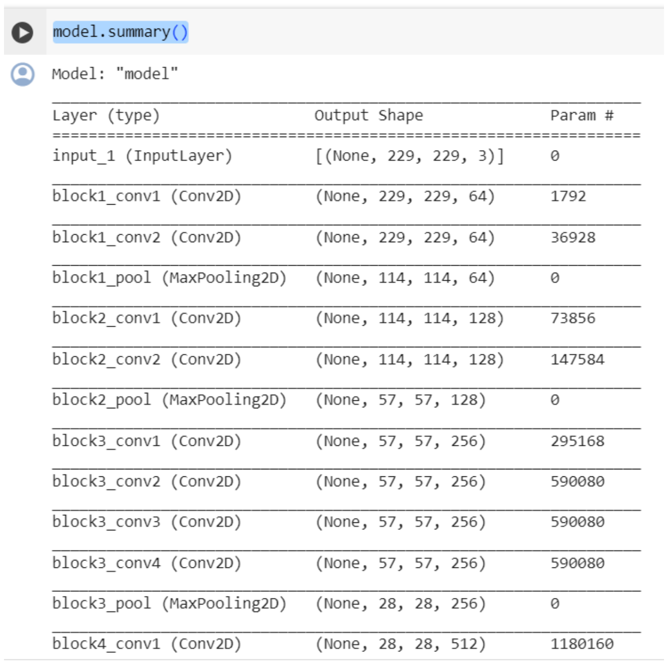
The final model needs to be compiled specifying the loss function (*categorical_crossentropy*) for multi-class classification, optimizer (*adam*), and evaluation metric (*accuracy*)[10]:

```
model.compile(
        loss='categorical_crossentropy',
        optimizer='adam',
        metrics=['accuracy']
)
```

The next step is to get the pre-trained weights for the VGG19 model from the TensorFlow website [11]. The message "Downloading data from..." indicates that the weights file is being fetched from the specified URL (https://storage.googleapis.com/tensorflow/keras-applications/vgg19/vgg19_weights_tf_dim_ordering_tf_kernels_notop.h5). For this, we'll download a file with .h5 extension called *vgg19_weights_tf_dim_ordering_tf_kernels_notop.h5*. The file contains the pre-trained weights for the VGG19 model without the top (fully connected) layers. This file is used for transfer learning with the VGG19 architecture, as it provides a starting point for training a new model on a different dataset (our own dataset) for predicting Alzheimer. Once the download is complete, the weights will be stored locally on our system and can be used to initialize the VGG19 model in our code. The line of code:

```
model.summary()
```

gives us the architecture of the model (Fig.3), with layers displayed in sequence from input to output, layer types, output shapes and the number of parameters in each layer.



```
model.summary()

Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 229, 229, 3)]     0
_________________________________________________________________
 block1_conv1 (Conv2D)       (None, 229, 229, 64)      1792
_________________________________________________________________
 block1_conv2 (Conv2D)       (None, 229, 229, 64)      36928
_________________________________________________________________
 block1_pool (MaxPooling2D)  (None, 114, 114, 64)      0
_________________________________________________________________
 block2_conv1 (Conv2D)       (None, 114, 114, 128)     73856
_________________________________________________________________
 block2_conv2 (Conv2D)       (None, 114, 114, 128)     147584
_________________________________________________________________
 block2_pool (MaxPooling2D)  (None, 57, 57, 128)       0
_________________________________________________________________
 block3_conv1 (Conv2D)       (None, 57, 57, 256)       295168
_________________________________________________________________
 block3_conv2 (Conv2D)       (None, 57, 57, 256)       590080
_________________________________________________________________
 block3_conv3 (Conv2D)       (None, 57, 57, 256)       590080
_________________________________________________________________
 block3_conv4 (Conv2D)       (None, 57, 57, 256)       590080
_________________________________________________________________
 block3_pool (MaxPooling2D)  (None, 28, 28, 256)       0
_________________________________________________________________
 block4_conv1 (Conv2D)       (None, 28, 28, 512)       1180160
```

Fig. 3. The architecture of the VGG19 model

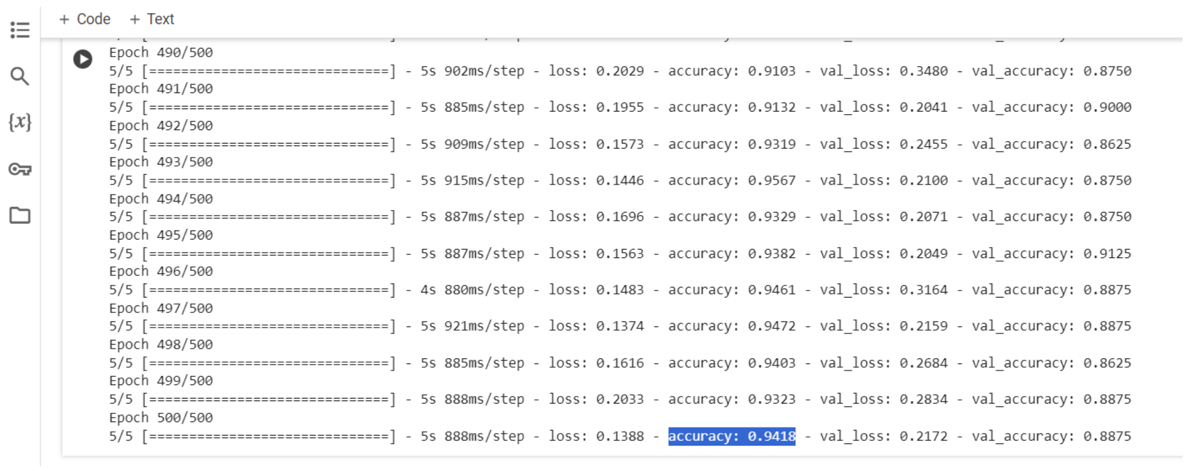
4 Training the model and predicting on the test data

The training will be made using the *fit* method (from Keras) with the observation that our training data is augmented on the fly and put it in the *train_aug* object [12]. This technique is used to artificially increase the size of our training dataset by applying various transformations (random transformations such as rotation, flipping and shifting) to the input data in order to improve the generalization and robustness of the trained model. The line of code for starting the training is:

```
history = model.fit(train_aug.flow(X_train, y_train, batch_size=batch_size),
                    validation_data=(X_test, y_test),
                    validation_steps=len(X_test) / batch_size,
                    steps_per_epoch=len(X_train) / batch_size,
                    epochs=epochs)
```

The *fit* method trains the model on the training data (*X_train* and *y_train*) using the specified parameters and evaluates its performance on the validation data (*X_test* and *y_test*, features and labels). It can be observed that the validation was done with test data (*X_test*) to which labels are known. In this way the results can be checked/validated. Test data it will be used again (only once) on the final model, which is completely trained with training data and validated with validation data sets. It should be specified that the test data was never seen by the model in the learning operation. During training, the model's weights are updated iteratively using an optimization algorithm (*Adam* optimizer [13]) to minimize the specified **loss** function (*categorical cross-entropy*). The training progress is monitored and recorded in the history object, which can be used for visualization and analysis after training is complete. The *train_aug* object contains the new training samples obtained after data augmentation by slightly modifying the existing ones. The flow operation generates batches of augmented training data (*X_train* and *y_train*) in real-time. The number of steps is calculated based on total number of images divided by the batch size (*epochs* = 500 and *batch_size* = 64).

During the training process, after each epoch (or a specified number of batches), the model's performance is evaluated on the validation data. This evaluation provides insights into how well the model is able to generalize to unseen data and helps identify whether the model is overfitting in the training process. The iterations are taking place over the entire training dataset. In figure 4, can be seen the progress of training and validation processes, together with the results: average loss over all training samples in the current batch (*loss*), the accuracy of the model on the current batch of training samples (*accuracy*), the average loss over all validation samples (*val_loss*) and the accuracy of the model on the current batch of validation samples (*val_accuracy*). In the figure below, it can be seen the final epoch (epoch 500) with the final training accuracy and the final validation accuracy: 0,94 and 0,88. These updates help monitor the performance of the model during training and provide insights into its learning behavior and convergence.



```
Epoch 490/500
5/5 [==============================] - 5s 902ms/step - loss: 0.2029 - accuracy: 0.9103 - val_loss: 0.3480 - val_accuracy: 0.8750
Epoch 491/500
5/5 [==============================] - 5s 885ms/step - loss: 0.1955 - accuracy: 0.9132 - val_loss: 0.2041 - val_accuracy: 0.9000
Epoch 492/500
5/5 [==============================] - 5s 909ms/step - loss: 0.1573 - accuracy: 0.9319 - val_loss: 0.2455 - val_accuracy: 0.8625
Epoch 493/500
5/5 [==============================] - 5s 915ms/step - loss: 0.1446 - accuracy: 0.9567 - val_loss: 0.2100 - val_accuracy: 0.8750
Epoch 494/500
5/5 [==============================] - 5s 887ms/step - loss: 0.1696 - accuracy: 0.9329 - val_loss: 0.2071 - val_accuracy: 0.8750
Epoch 495/500
5/5 [==============================] - 5s 887ms/step - loss: 0.1563 - accuracy: 0.9382 - val_loss: 0.2049 - val_accuracy: 0.9125
Epoch 496/500
5/5 [==============================] - 4s 880ms/step - loss: 0.1483 - accuracy: 0.9461 - val_loss: 0.3164 - val_accuracy: 0.8875
Epoch 497/500
5/5 [==============================] - 5s 921ms/step - loss: 0.1374 - accuracy: 0.9472 - val_loss: 0.2159 - val_accuracy: 0.8875
Epoch 498/500
5/5 [==============================] - 5s 885ms/step - loss: 0.1616 - accuracy: 0.9403 - val_loss: 0.2684 - val_accuracy: 0.8625
Epoch 499/500
5/5 [==============================] - 5s 888ms/step - loss: 0.2033 - accuracy: 0.9323 - val_loss: 0.2834 - val_accuracy: 0.8875
Epoch 500/500
5/5 [==============================] - 5s 888ms/step - loss: 0.1388 - accuracy: 0.9418 - val_loss: 0.2172 - val_accuracy: 0.8875
```

Fig. 4. The training and validation processes

Using the function *save* in Keras we are now saving the entire model (architecture, weights, and training configuration) to a file in the Hierarchical Data Format 5 (HDF5) format. This allows us to store the trained model for later use, deployment, or sharing with others. Here's how the function works [4]:

```
model.save('AD_Model_VGG19.h5')
```

We can then load this saved model using the *keras.models.load_model()* function to reuse it for inference, evaluation or further training:

```
model = load_model('AD_Model_VGG19.h5')
from tensorflow.keras.models import Model, load_model
```

Please note that we were able to load this model because we imported initially, at the start of this programme, the *load_model* function from *tensorflow.keras.models*.
Having the model (already trained), we are ready to use it and obtain predictions from it on the test data (X_test) [14]. It's better to specify the batch size to help control memory usage and computation time:

```
y_pred = model.predict(X_test, batch_size=batch_size)
```

After executing this line of code, *y_pred* will contain the predicted outputs (or probabilities) for each sample in the test dataset. These predictions can then be further analyzed, evaluated, or compared with the ground truth labels (*y_test*) to assess the performance of the model.
Our next intention is to visualize the predictions made by our model for a subset of the test data (the first 9 samples in the test dataset). We are therefore looping through the

selected predictions, display the corresponding images and add titles to the plots indicating the predicted probabilities of being Demented or NonDemented**:**

```
prediction=y_pred[1:10]
for index, probability in enumerate(prediction):
  if probability.item(0) > 0.5:
        plt.title('%.2f' % (probability.item(0)*100) + '% Demented')
  else:
        plt.title('%.2f' % ((1-probability.item(0))*100) + '% NonDemented')
  plt.style.reload_library
  plt.imshow(Dem_images[index])
  plt.show()
```

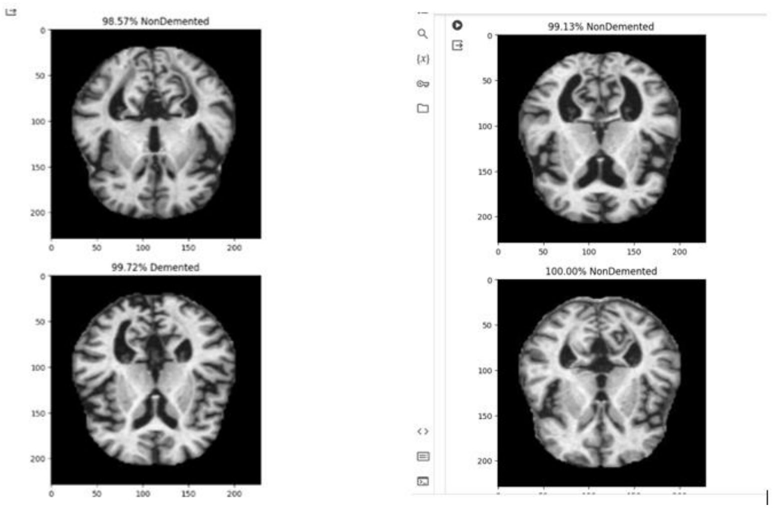


Fig. 5. The predictions

In figure 5 is presented the visualization of predictions.

5 Various visualizations and graphs

The ROC curve illustrates (as in figure 6) the performance of our binary classification model [15]. A perfect classifier would have a curve that goes straight up to the top-left corner: 100% sensitivity (true positive rate) and 100% specificity. The diagonal line from the bottom-left to the top-right reflects random guess. In general, the closer the ROC curve is to the top-left corner, the better the classifier is at distinguishing between the two classes. The code for ROC curve visualization is:

```
y_pred_bin,y_test_bin=None,None
y_pred_bin = np.argmax(y_pred, axis=1)
y_test_bin = np.argmax(y_test, axis=1)

fpr, tpr, thresholds = roc_curve(y_test_bin, y_pred_bin)
auc = roc_auc_score(y_test_bin, y_pred_bin)
print('AUC: %.3f' % auc)
plt.plot(fpr, tpr)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.0])
plt.rcParams['font.size'] = 12
plt.title('ROC curve for our model')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.savefig('VGG19_ROC.png')
plt.grid(True)
```

In figure 6 can be seen the resultant ROC curve**:**

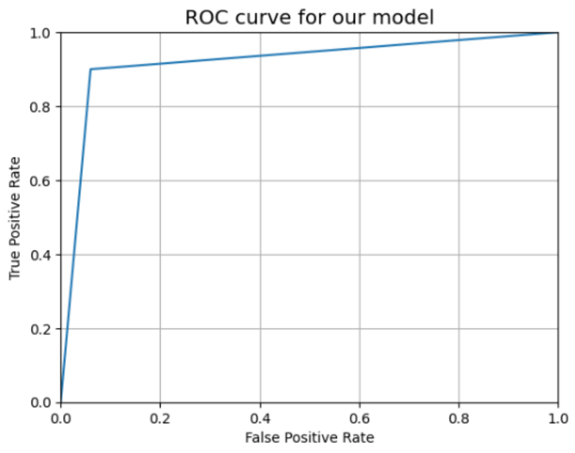


Fig. 6. The ROC curve

In order to plot a Confusion Matrix, we'll define a function called *plot_confusion_matrix* and and we'll use the *confusion_matrix* function with the following arguments:

- *y_test_bin* contains the true labels,
- *y_pred_bin* contains the predicted labels.

The normalize parameter determines whether to normalize the confusion matrix or not, based on the value of the normalize argument passed to the function:

```
def plot_confusion_matrix(normalize):
  classes = ['DEM','NONDEM']
  cn = confusion_matrix(y_test_bin, y_pred_bin,normalize=normalize)
```

We intend to plot the confusion matrix as a heatmap using the seaborn library. We'll save the plot as an image file named 'VGG19_Confusion_Matrix.png', and we'll display the plot on the screen (figure 7):

```
sns.heatmap(cn,cmap='plasma',annot=True)
  plt.xticks(tick_marks, classes)
  plt.yticks(tick_marks, classes)
  plt.title('Confusion Matrix')
  plt.ylabel('True label')
  plt.xlabel('Predicted label')
  plt.savefig('VGG19_Confusion_Matrix.png')
  plt.show()
print ('Confusion Matrix fara Normalizare')
plot_confusion_matrix(normalize=None)

print('Confusion Matrix cu Normalizare')
plot_confusion_matrix(normalize='true')
```

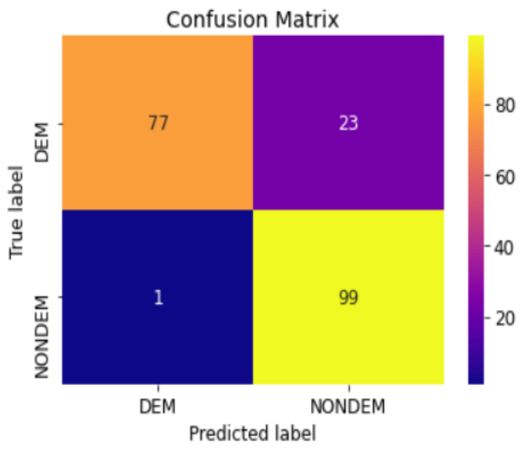


Fig. 7. Confusion Matrix without normalization

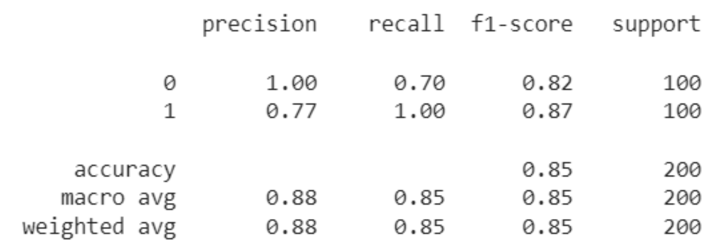
Reading the matrix:

- The model had 77 good predictions for Demented samples (True positive) from 78 samples (where 1 was a wrong prediction of being Demented (False positive).
- The model predicted wrong 23 samples as being Demented (False positive) and has good prediction of 99 samples of being non-demented (True negative) from 122. The total samples of positive class (Dem) is 78 and the total samples for negative class (NonDem) is 122. The total of samples (test set) is 200.

Applying the formula for the Accuracy from the Confusion Matrix, accuracy is the number of samples correctly classified out of all the samples present in the test set: (77+99)/(77+23+1+99)= 176/200= **0.88.**

Using the *classification_report* function from *scikit-learn* we are generating a comprehensive report of the classification performance, including precision, recall, F1-score, and support for each class (Figure 8):

```
from sklearn.metrics import classification_report
print(classification_report(y_test_bin, y_pred_bin))
```

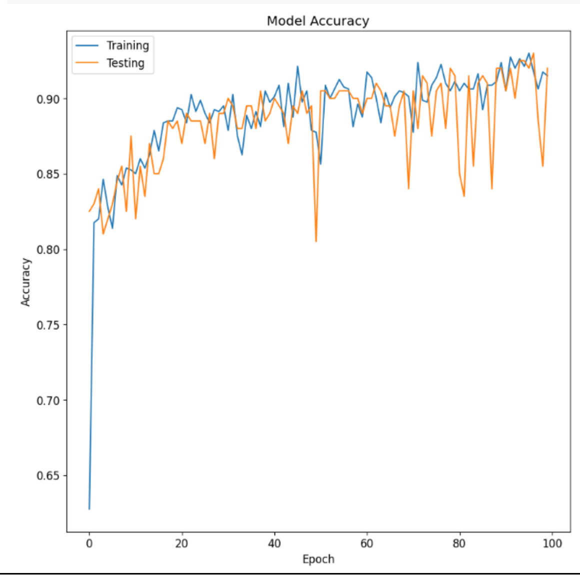


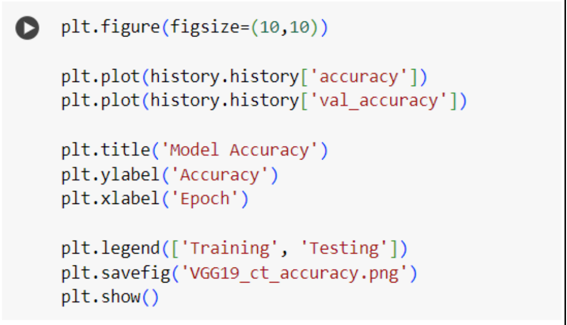
```
              precision    recall  f1-score   support

           0       1.00      0.70      0.82       100
           1       0.77      1.00      0.87       100

    accuracy                           0.85       200
   macro avg       0.88      0.85      0.85       200
weighted avg       0.88      0.85      0.85       200
```

Fig. 8. The Classification report

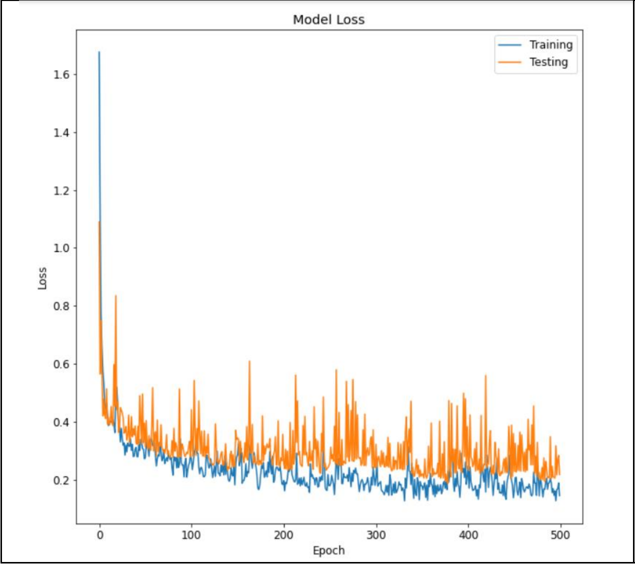
Continuing to visualize the results based on the information captured in the *history* object, in the table 1 the graphs are drawn for the training and testing *accuracy*, the training and testing *loss* of the neural network used here. The loss quantifies the difference between the predicted output of the model and the actual target values. The goal of training a machine learning model is to minimize this loss, which essentially means making the model's predictions as close to the actual target values as possible.

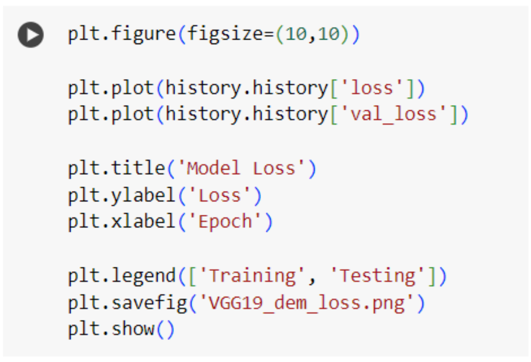
Table 1. Graphs of accuracy and loss evolution through epochs





```
plt.figure(figsize=(10,10))

plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])

plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')

plt.legend(['Training', 'Testing'])
plt.savefig('VGG19_ct_accuracy.png')
plt.show()
```





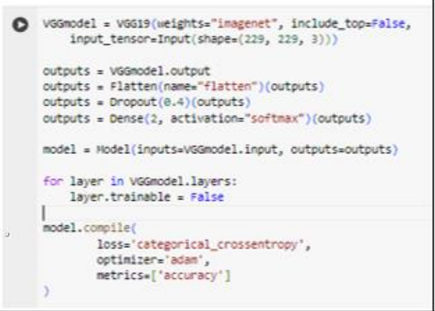
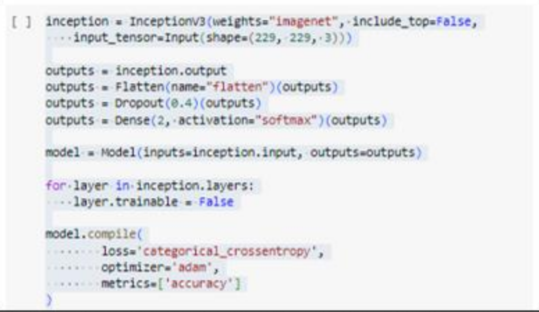
```
plt.figure(figsize=(10,10))

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])

plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')

plt.legend(['Training', 'Testing'])
plt.savefig('VGG19_dem_loss.png')
plt.show()
```

6 VGG19 versus Inception_V3

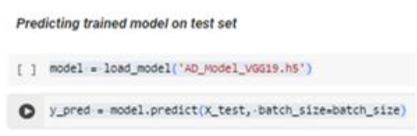
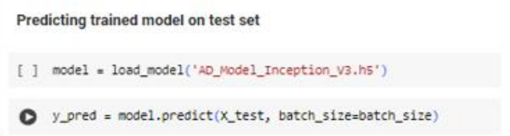
In the above experiment it was seen how, by using a pre-trained model (which involves training the model on a large dataset such as ImageNet (containing millions of labeled images), the results are impressive. We now propose to choose another model that has been widely adopted in the computer vision community for its efficiency and effectiveness in various tasks: Inception_v3.

Table 2. VGG19 versus Inception_v3

Task name and results	VGG10 Model	Inception_V3 Model
Importing libraries	from tensorflow.keras.applications import VGG19	from tensorflow.keras.applications.inception_v3 import InceptionV3

Building the model	
	`VGGmodel = VGG19(weights="imagenet", include_top=False,`
	`    input_tensor=Input(shape=(229, 229, 3)))`
	
	`outputs = VGGmodel.output`
	`outputs = Flatten(name="flatten")(outputs)`
	`outputs = Dropout(0.4)(outputs)`
	`outputs = Dense(2, activation="softmax")(outputs)`
	
	`model = Model(inputs=VGGmodel.input, outputs=outputs)`
	
	`for layer in VGGmodel.layers:`
	`    layer.trainable = False`
	
	`model.compile(`
	`        loss='categorical_crossentropy',`
	`        optimizer='adam',`
	`        metrics=['accuracy']`
	`)`
		
		`inception = InceptionV3(weights="imagenet", include_top=False,`
		`    input_tensor=Input(shape=(229, 229, 3)))`
		
		`outputs = inception.output`
		`outputs = Flatten(name="flatten")(outputs)`
		`outputs = Dropout(0.4)(outputs)`
		`outputs = Dense(2, activation="softmax")(outputs)`
		
		`model = Model(inputs=inception.input, outputs=outputs)`
		
		`for layer in inception.layers:`
		`    layer.trainable = False`
		
		`model.compile(`
		`        loss='categorical_crossentropy',`
		`        optimizer='adam',`
		`        metrics=['accuracy']`
		`)`

URL path to Google Cloud Storage for downloading the pre-trained set of weights	*https://storage.googleapis.com/tensorflow/keras-applications/vgg19/vgg19_weights_tf_dim_ordering_tf_kernels_notop.h5*	*https://storage.googleapis.com/tensorflow/keras-applications/inception_v3/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5*

Predicting	
	Predicting trained model on test set
	`[ ] model = load_model('AD_Model_VGG19.h5')`
	`y_pred = model.predict(X_test, batch_size=batch_size)`
		
		Predicting trained model on test set
		`[ ] model = load_model('AD_Model_Inception_V3.h5')`
		`y_pred = model.predict(X_test, batch_size=batch_size)`

Training accuracy	0.94	0.93
Validation accuracy	0.88	0.95

One of the distinctive features of the Inception architecture is the use of inception modules which are modules that perform parallel convolutions at different scales and concatenate their results. These modules help capture features at different levels of abstraction efficiently. Due to its pre-trained weights on large datasets like ImageNet, Inception_v3 can be utilized for transfer learning, the same we did previously with VGG19. Transfer learning involves taking a pre-trained model and fine-tuning it on a smaller, domain-specific dataset (in our case was a set of RMN images) to adapt it to a particular task. We will apply the same code but with some changes which are reflected in the Table 2. After running the new code involving Inception_v3 pre-trained model, the results are even better: the validation accuracy is higher than when we used VGG19. Important is to understand the concept of using these 2 models, to understand how and why use the pre-trained weights (without the weights from the fully connected layers on top) which are downloaded from Google Cloud Storage. The names of these files have the *.h5* extension which indicates that it's stored in the Hierarchical Data Format version 5 (HDF5), a common format for storing large numerical datasets. Also, the names of these files containing the pre-trained weights, have "*notop*" indicative pointing clearly the fact that the top fully connected layers were eliminated:

- `vgg19_weights_tf_dim_ordering_tf_kernels_notop.h5`
- `inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5`.

7 State of the Art

A couple of years ago, 2 researchers from Imperial College London and from the Department of Biomedical Engineering, made their own experiences in predicting AD using 3D images and 3D CNN's. As opposed to 2D images, 3D images contain volumetric data, capturing spatial structures and relationships in three dimensions [20]. Together with 3D CNN's, these three-dimensional data capture spatiotemporal patterns providing richer information about the shape, size and spatial distribution of AD phenomena. While 2D images used in this work offer simplicity and ease of interpretation, 3D images provide richer spatial information and may lead to improved accuracy and performance in tasks such as object recognition, segmentation and tracking, as they capture a more complete representation of the underlying 3D scene. Going back to the experiences of the UK researchers, they obtained an accuracy of 95,39% using a combination of sparse autoencoders and convolutional neural networks. Their approach using 3D images and 3D convolutions (on the whole MRI image) yielded better performance. However, 3D brain MRI is not available in most clinical settings because 3D MRI sequences have longer acquisition times than two-dimensional (2D) MRI, which significantly increase computational burden, storage, and cost. Thus, the DL algorithms for diagnosing AD using 3D MRI may be difficult to apply to most brain MRI scans obtained in typical clinical settings.

In 2021, a group of researchers from Bangladesh and India, presented a comparative analysis of Machine Learning algorithms to predict Alzheimer's disease [21]. Even though Deep Learning algorithms have been used in this paper describing our experiments, both ML and DL can be used for predicting dementia and the choice between choosing ML and DL depends on factors such as the size and complexity of the dataset, the availability of labelled data, computational resources and the expertise of the practitioners. In ML models, features or attributes used for training are manually designed and engineered by domain experts rather than being automatically learned from the data by the model itself, as in DL models. In their research, the engineers concluded that, with their particular dataset (provided by the Open Access Series of Imaging Studies) the system got the best results using Support Vector Machine (SVM) as opposed to others ML algorithms such as logistic regression, decision tree and random forest. Their comparison table of models is presented in Table 3.

Table 3. Comparison table of ML models

Model	Accuracy (%)	Recall (%)	Precision (%)	AUC (%)	*F*1 score
SVM	92.0	91.9	91.9	91.9	91.9%
Logistic regression	74.7	70.3	76.5	74.6	73%.3
Decision tree	80.0	59.4	100	79.7	74.5%
Random forest	81.3	70.3	84.4	81.2	76.7%

The study called "Generalizable deep learning model for early Alzheimer's disease detection from structural MRIs" introduces a groundbreaking AI-driven approach for early Alzheimer's disease detection, leveraging 3D deep convolutional neural networks on structural MRIs [22]. The model showcases exceptional accuracy, with an impressive area-under-the-curve (AUC) of 85.12% when distinguishing between cognitively normal subjects and those with mild cognitive impairment (MCI) or mild Alzheimer's dementia. Additionally, it achieves an AUC of 62.45% in detecting MCI, a task known for its complexity.
Its utilization of advanced AI techniques not only enhances diagnostic precision but also significantly expedites the process compared to conventional methods. Furthermore, the AI model demonstrates promising capabilities in predicting disease progression, leveraging its ability to autonomously learn and identify imaging biomarkers associated with Alzheimer's pathology. These remarkable findings underscore the transformative potential of AI in revolutionizing early diagnosis and facilitating more effective clinical interventions for Alzheimer's disease.
Another study happened in Korea called "*Deep learning-based diagnosis of Alzheimer's disease using brain magnetic resonance images***"** investigated the potential of a Korean-home-made convolutional neural network (called VUNO Med-DeepBrain AD) for diagnosing AD using 2D brain MR images as input data (using T1-weighted magnetic resonance imaging which is often used for brain imaging) [23]. The research involved 98 elderly participants aged 60 years or older from Seoul Asian Medical Centre and the Korea Veterans Health Service. Results showed accuracy of 87.1%. In their experiments, they used Inception-v4, as opposed to Inception-v3 employed in our experiments described in this paper. The Inception-v4 has more modules being the next generation of Inception-v3 pretrained models. In the end, their work and their results suggested that VUNO and his algorithm could serve as a valuable tool for supporting clinical decisions, particularly for physicians not specialized in Alzheimer disease, thereby improving the accessibility of AD diagnosis and treatment.

7 Conclusions

In this project, our specific tasks were Alzheimer's disease prediction. Instead of building models from scratch, we leveraged pretrained models and transferred the knowledge learned from huge datasets. The pre-trained models served as powerful features extractors: the early layers of these models learn to detect low-level features like edges and textures while deeper layers learn more abstract features like shapes and patterns. The top layers (which are typically fully connected layers) are fine-tuned with new training data (as we saw in our experiments) while the early layers (convolutional and pooling layers) are frozen, their weights were not updated retaining the learned representation. More than that, these models alleviate some of the computational resources burden because we only needed to fine-tune the models to our specific task with fewer resources and smaller dataset. When working with models like VGG19 and Inception_v3, we were able to find extensive documentation, tutorials and support in the deep learning community. However, using pretrained models, the fine-tuning requires careful hyperparameters tuning and experimentation.

References

[1] A. Moscoso and J. Silva, "Prediction of Alzheimer's disease dementia with MRI beyond the short-term," *National Library of Medicine,* 2019.
[2] A. Md. Murshid, B. Rejwana Parvin and I. Ariful, "Unveiling Neuroprotective Potential of Spice Plant-Derived Compounds

against Alzheimer’s Disease: Insights from Computational Studies," *International Journal of Alzheimer's Disease.*

[3] T. M., R. B. and K. R. U., "Machine Learning Techniques for the Diagnosis of Alzheimer’s Disease: A Review," *ACM Transactions on Multimedia Computing, Communications, and Applications,* vol. 16, no. 1s, pp. 1-35, 2020.

[4] M. Lagunas and E. Garces, "Transfer Learning for Illustration Classification," in *CEIG - Spanish Computer Graphics Conference* , Spanish, 2017.

[5] . J.-K. Joanna, K. Pawel and G. Marek , "Melanoma Thickness Prediction Based on Convolutional Neural Network with VGG-19 Model Transfer Learning," Poland.

[6] "Keras documentation," [Online]. Available: https://keras.io/api/applications/vgg/.

[7] C. P, M. H and . V. N, "A review of medical image data augmentation techniques for deep learning applications," *The Journal of Medical Imaging and Radiation Oncology (JMIRO)* , 2021.

[8] A. Samer , . G. Tom and R. Scott , "Artificial intelligence and machine learning overview in pathology & laboratory medicine: A general review of data preprocessing and basic supervised concepts," *Seminars in Diagnostic Pathology,* vol. 40, pp. 78-87, 2023.

[9] "Scikit-learn documentation," [Online]. Available: https://scikit-learn.org/stable/modules/model_evaluation.html.

[10] "Matplotlib documentation," [Online]. Available: https://matplotlib.org/stable/tutorials/images.html.

[11] "Normalization in image preprocessing," [Online]. Available: https://medium.com/@patelharsh7458/normalization-in-image-preprocessing-scaling-pixel-values-by-1-255-111b2fa496d4.

[12] "One hot encoding in Machine Learning," [Online]. Available: https://www.geeksforgeeks.org/ml-one-hot-encoding-of-datasets-in-python/.

[13] "Transfer learning," [Online]. Available: https://www.learndatasci.com/tutorials/hands-on-transfer-learning-keras/.

[14] "Loss function in Machine Learning," [Online]. Available: https://www.datacamp.com/tutorial/loss-function-in-machine-learning.

[15] "Tensor Flow," [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/applications/vgg19/VGG19.

[16] "Image augmentation," [Online]. Available: https://d2l.ai/chapter_computer-vision/image-augmentation.html.

[17] "Adam optimizer," [Online]. Available: https://towardsdatascience.com/the-math-behind-adam-optimizer-c41407efe59b.

[18] "Keras Predict method," [Online]. Available: https://keras.io/api/models/model_training_apis/.

[19] F. S. Nahm, "Receiver operating characteristic curve: overview and practical use for clinicians," *Korean Journal of Anesthesiology,* vol. 75, pp. 25-36, 2022.

[20] . S. A. Sarah and E.-D. A. El-Sayed , "Predicting Alzheimer's Disease with 3D Convolutional Neural Networks," *International Journal of Applications of Fuzzy Sets and Artificial Intelligence,* vol. 1, 2020.

[21] A. B. Morshedul, J. S. A H M , . M. Maliha and . K. M. Mohammad , "A Comparative Analysis of Machine Learning Algorithms to Predict Alzheimer's Disease," 2021.

[22] L. Sheng , . M. V. Arjun and R. Henry , "Generalizable deep learning model for early Alzheimer's disease detection from structural MRIs," 2022.

[23] K. S. Jun , . H. . W. Ji , B. B. Jong , . M. G. Dong and . S. Jin , "Deep learning-based diagnosis of Alzheimer’s disease using brain magnetic resonance images: an empirical study," 2022.

[24] "Keras documentation," [Online]. Available: https://keras.io/api/models/model_saving_apis/model_saving_and_loading/.

[25] "Understanding Confusion Matrix," [Online]. Available: https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62.

Paul TEODORESCU is an Engineer with an international background in engineering and IT. He has worked in IT field in Romania and Canada. Specializing in databases, PL/SQL, Oracle, Data Warehousing, Business Intelligence, Artificial Intelligence (Machine Learning, Artificial Neural Networks, Natural Language Processing), he studied and worked for 11 years in Canada. He is currently working at Computer Science Research Institute in Bucharest - ICI - as a research scientist and is involved in Artificial Intelligence, NLP and GIS projects.

Silvia OVREIU is PhD in Deep Learning at the Faculty of Electronics, Telecommunications and Information Technology at the University Politehnica of Bucharest, with the thesis titled "Retinal Image Analysis using Deep Learning Algorithms." She was involved in research within the UPB Proof of Concept 2020 Project and served as a research assistant. She was the principal investigator of the SAIGHT project (Software for Automatic Analysis of Ocular Images). The main goal of the project is to create a platform based on Deep Learning for the automatic analysis and diagnosis of ocular diseases such as glaucoma. She is involved in organizing the International Summer School on Imaging with Medical Applications (SSIMA).

Mădălina ZAMFIR graduated from the Faculty of Automation and Computers at the Polytechnic University of Bucharest. She is scientific researcher within the *Digital Transformation and Governance* Department at the National Institute for Research and Development in Informatics – ICI Bucharest. Topics of interest in the research activity cover Cloud infrastructures, IoT support for Big Data, Big Data Analytics, Geospatial Analytics, Metaverse.

Cristian ȚÎRLEA is a graduate of the Faculty of Electrical Engineering at the Politehnica University of Bucharest. He is currently a student at the Informatics faculty of the University of Bucharest. He is currently working at Computer Science Research Institute in Bucharest - ICI - as a research assistant and is involved in Artificial Intelligence. He is involved in Advanced Artificial Intelligence Techniques in Science and Applied Fields project, which aims to develop an advanced model for predicting neurological outcomes based on neuroimaging and clinical data, using machine learning algorithms to improve diagnosis and prognosis in various conditions neurological.