NOVA IMS Information Management School

MAA

Mestrado em Métodos Analíticos Avançados
Master Program in Advanced Analytics

Convolutional Neural Networks in Alzheimer's Classification

Hugo Fernandes - 20210682

Martim Figueira - 20210686

Nuno Penim - 20210998

Paulo Oliveira - 20211002

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

INTRODUCTION

With our project, we aim in developing a model capable of identifying the presence and stage of Alzheimer's disease in medical MRI scans. We believe that this topic is a great starting point, as currently more and more people suffer from Alzheimer's or other form of dementia as they age.

Having an automated system, could be beneficial to lower the strain on medical personnel as well as speeding up the detection process of this disease. We will develop a Neural Network of the type CNN.

Our choice in using a Convolutional Neural Network (CNN) is due to the fact that these types of Neural Networks are better in image classifications and medical image analysis - which is the main goal of our project - than RNN, which are better for audio and handwriting classification, for example.

TASK DEFINITION

Our system is aimed at the early automated detection of the Alzheimer's disease in medical patients. Currently this diagnosis is made by a doctor, observing an MRI scan of the brain of the patient. There are more and more systems that do this task automatically, however we aim in using all the lectured subjects and capabilities we acquired while studying Deep Learning, as well as trying to mix various algorithms to check for detection errors and presenting a result that is closer to the truth.

We don't believe the scope of the project is too narrow, as the Alzheimer's disease is a multistage disease, possibly to be described in 4 general stages, non-demented, very mild demented, mild demented and moderate demented, or too broad, as we are only working in the detection of the Alzheimer's disease. With the development of the project, we will be able to judge this scope better, and in the end, we will present a conclusion about it.

Our system will receive the MRI scan of a patient, and after applying deep learning techniques, it will return if Alzheimer has been detected on the patient, and at which stage of the disease.

In the evaluation step of our project, we decided to compare with another notebook that is present in the Kaggle platform. This notebook was the highest rated one that seemed to present an original solution. The next notebooks claimed to be a copy with improvements, and after a check, the few changes made were mostly in the network architecture itself, and not in the preprocessing or any other step. In the approach part of this report, it is explained that the preprocessing follows a similar procedure, since all the other solutions also seemed to follow this procedure. We however changed it, as we did not consider it to be correct.

APPROACH

Since this was an image classification problem, we decided in using CNN, since as lectured in class this was the most effective method in classifying images.

Initially, we started by observing the dataset. The dataset was already divided in train and test and it was already categorized in the different stages of Alzheimer's disease. Since there were other solutions to this problem, we decided to observe the approach taken by them, to base our own preprocessing. Therefore, we mixed the train and test folders, and kept the classes. Like this we can add randomness into our model, to know how to classify each.

After reclassifying our data, we decided to balance the dataset, using a SMOTE technique. Here is where our preprocessing differs from the other Deep Learning CNN specific solution present online. This solution had a very high validation accuracy, of 98%. This seemed very difficult to us to obtain, and we considered initially there was a mistake either in the preprocessing or in the model implementation itself. This solution applies the SMOTE technique before performing a train-test split. In our opinion, this shouldn't be done, as the test should be done with real data, and not created data. Considering this, we applied our train-test split, with a test size of 20%, then performed SMOTE on our train dataset, in order to balance the data.

The next step was to run our Dataset through a CNN. As a base to our CNN model, we used the class example. Before applying SMOTE, the model had a validation accuracy of 32%, which obviously was not acceptable. After applying SMOTE, the execution of the model yielded 63%. While this was better than 32%, we believed we could do much better, by tweaking. The first thing we did was adding an extra layer. In

the class example, the model started at dimensionality 32, so we put this layer before, at 16. This did not change much, although the validation accuracy jumped to 68%. After researching the options available in the Keras documentation, we suggested introducing the option "padding", which would make all the outputs equal as the inputs. This small change increased our final accuracy to 70%. The last step in the model conception was adding more Dropout layers. We had a single dropout layer before the final layer, with a 50% dropout rate. We decided to add 3 dropout layers after the 16, 32 and 64 dimensionality layers, all of them with a dropout rate of 10%. This change increased our accuracy to 78%. After playing more with the dropout layer values, we did not obtain better results than this, so in the model architecture, this was the best accuracy we could obtain.

Lastly, we implemented a scheduler. In the previous step we were constantly running with 10 epochs, obtaining a maximum of 78% in validation accuracy. With a scheduler implemented, our model ran for longer, but yielded better scores, with a validation accuracy of 87% and a training accuracy of 95%. This showed us there was not significant overfitting in our model, and our model was categorizing 87% of the test images in their correct category.

EVALUATION METRICS

During the execution, we did not notice a noticeable increase in system memory usage, however in early iterations the program would take a lot of memory. This happened due to the fact the batch size and epoch number was unadjusted. After a correction to the correct parameters, there was no noticeable memory usage. With this in mind, in our hardware, the total run time of our program was of 30 minutes and 41 seconds. We obtained 95% accuracy in the train dataset and 87% of accuracy in the validation dataset.

```
Epoch 26/100
255/255 [==============================] - 51s 199ms/step - loss: 0.0952 - acc: 0.9636 - val_loss: 0.5047 - val_acc: 0.8301
Epoch 27/100
255/255 [==============================] - 51s 199ms/step - loss: 0.1009 - acc: 0.9619 - val_loss: 0.3895 - val_acc: 0.8652
Epoch 28/100
255/255 [==============================] - 51s 199ms/step - loss: 0.1155 - acc: 0.9586 - val_loss: 0.3749 - val_acc: 0.8643
Epoch 29/100
255/255 [==============================] - 51s 198ms/step - loss: 0.0894 - acc: 0.9674 - val_loss: 0.3977 - val_acc: 0.8613
Epoch 30/100
255/255 [==============================] - 51s 198ms/step - loss: 0.1049 - acc: 0.9616 - val_loss: 0.3960 - val_acc: 0.8711
Epoch 31/100
255/255 [==============================] - 51s 199ms/step - loss: 0.0891 - acc: 0.9684 - val_loss: 0.4218 - val_acc: 0.8604
Epoch 32/100
255/255 [==============================] - 51s 198ms/step - loss: 0.0917 - acc: 0.9675 - val_loss: 0.4980 - val_acc: 0.8398
Epoch 33/100
255/255 [==============================] - 51s 198ms/step - loss: 0.0830 - acc: 0.9712 - val_loss: 0.5063 - val_acc: 0.8311
Epoch 34/100
255/255 [==============================] - 51s 199ms/step - loss: 0.0854 - acc: 0.9682 - val_loss: 0.4678 - val_acc: 0.8555
Epoch 35/100
255/255 [==============================] - 51s 198ms/step - loss: 0.0886 - acc: 0.9673 - val_loss: 0.4560 - val_acc: 0.8438
Epoch 36/100
255/255 [==============================] - 51s 199ms/step - loss: 0.0854 - acc: 0.9675 - val_loss: 0.3848 - val_acc: 0.8779
```

Figure 2: Last 10 epochs of our training execution

On the model available in the Kaggle platform, we do not consider the accuracy of the validation valid, since as we mentioned before, the validation portion of the dataset contained SMOTE images, which we do not consider like real images the dataset could have. Still, while running for more epochs than our model, the other similar solution ran for a total of 16 minutes. This might have been due to better hardware, as the average epoch run time for our model on our hardware was of 51 seconds, and the average run time of the Kaggle model was of 15 seconds. The Accuracy of the train dataset of this implementation was of 99% and the validation accuracy was of 98%. The model itself differed too. The result with testing data was of 82% in the model in the Kaggle platform. On the other hand, our model had a test accuracy result of 88%, with the real test images.

```
model.evaluate(test_data,test_labels)
```

```
40/40 [==============================] - 2s 39ms/step - loss: 0.3774 - acc: 0.8813
```

Figure 1: Result of our model, using the test dataset

We believe most of our results come from our optimized preprocessing, since our validation is also being made with real pictures, and not SMOTE results. Like this, our validation accuracy is closer to our test result accuracy, so we could estimate what the result would be while still developing the model. Still, while we think most of the better results of the model came from the preprocessing, our model had a different architecture than the model present on Kaggle, which could have been beneficial to us.

ERRORS

After the model was complete, we decided to collect data on the model's performance and perform a few tests, to understand better our network, and to see how the model itself behaved.

Initially, in the performance data collection stage, we collected data on the training and validation accuracy readings. We used the model history function and plotted a graph, using Python's plotly library. The graph is presented below.

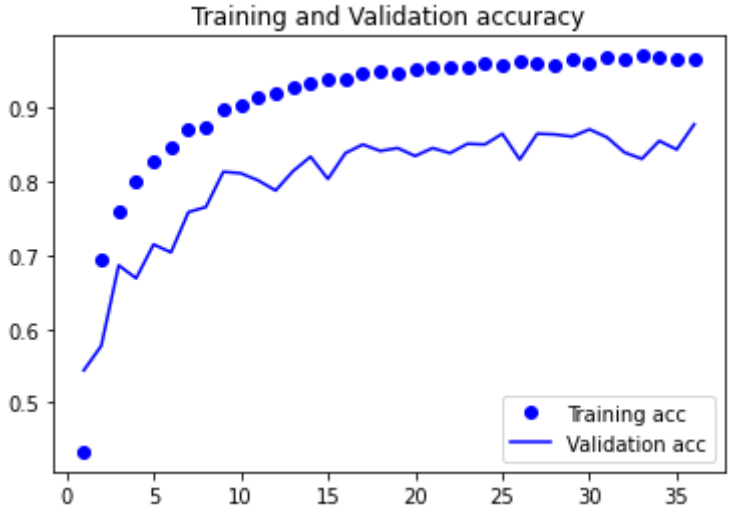

Figure 3: training and validation accuracy

From the graph observation, both the training and validation accuracy started low, eventually stagnating at a certain value, that we considered the maximum. For the training dataset this value was of a 96% accuracy, and for the validation dataset of about 87%. Generally, in an overfitting situation, there is a big gap between the training dataset accuracy and the validation dataset accuracy. A drastic example could be the training having 99% accuracy and the validation having about 50%. While our model is not perfect, we do not consider that it presents significant overfitting.

The next variable that we took in consideration was the loss of the model. This variable shows how the network is performing, diagnosing the type of errors the model is making. Generally, low loss and low accuracy indicates a lot of small errors, low loss and high accuracy indicates a few small errors, high loss and low accuracy indicate a lot of big errors and high loss and high accuracy indicate a few big errors. Bellow, there's a graph of the loss of our model.

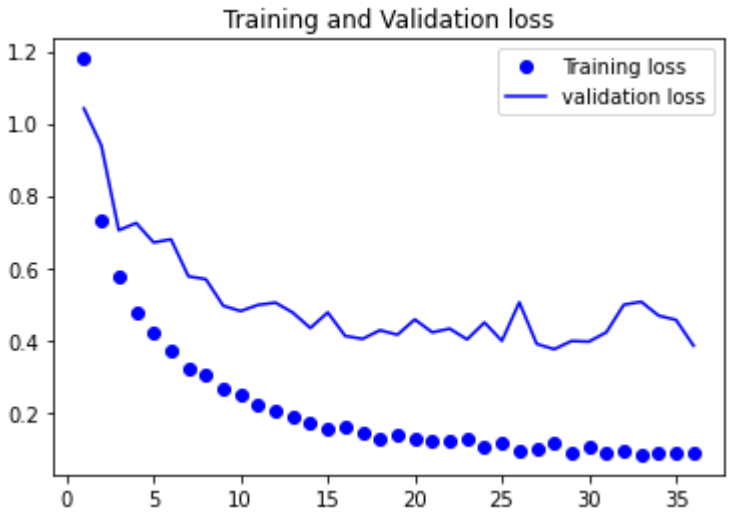

Figure 4: Training and validation loss

From observing the graph, the loss of both the training and validation also tends to stabilize at a certain value. In the training dataset context, this value could be as low as 0.008, which associated with its 96% accuracy shows that the model is making a few small errors. In the validation dataset case, the loss value is around 0.38. Associated with the 87% of validation accuracy, we can conclude that the model is making a few big mistakes in categorizing the data. It's important to mention that the testing dataset behaves similarly to the validation dataset, since as we mentioned before, both the validation and testing datasets are not SMOTEd, being both with real world data.

After this, we decided in performing a small test, in order to see the generalization ability of the model. This test consisted of a sequence of executions of the code, in order to randomize the datasets, as they are selected by splitting them randomly into training, validation and testing. Since the training of the model takes around 25 minutes per run, we did a total of 10 runs. Below, there's a table with the data of the accuracy and loss values of the testing, which in all runs presented a similar value to the validation dataset.

Run	Accuracy	Loss
1	0.86	0.38
2	0.83	0.42
3	0.88	0.36
4	0.89	0.36
5	0.87	0.4
6	0.85	0.39
7	0.88	0.36
8	0.87	0.35
9	0.88	0.36
10	0.89	0.36
Avg	0.87	0.374

Table 1: Loss and Accuracy of our test runs

From a brief analysis of the table, we can see that the average result stayed close to our original accuracy result (87% vs 88%). The average loss value was the same as the value of the original run, 0.37. This indicates that our model, in our hardware, is able to maintain consistency in the data.

As final results, we will consider 87% of accuracy and 0.37 of loss. These values indicate that our model is capable of accurately classifying MRI scans of Alzheimer's patients. For comparison, the Notebook available in the Kaggle platform used for our metrics had 82% of accuracy and a loss of 1.5. This means that our network is more accurate, and has smaller mistakes than the solution present online.

CONCLUSION

With this project, we were able to build a CNN model capable of accurately identifying the present and/or stage of Alzheimer's disease in an MRI scan image of a possible patient. While we are happy with the accuracy value of our result, and even while the loss value is lower than the existing solution, we believe there could have been tricks we did not discover or explore, to make it possible.

To lower the general loss, we tried adding more dropout layers, changed the number of the layers and changing the activation functions, however the obtained results were either not good enough, or we did not consider them to be worthy, resulting in these changes being reverted.

REFERENCES

- The Notebook used for comparison: https://www.kaggle.com/code/amyjang/alzheimer-mri-model-tensorflow-2-3-data-loading
- Tensorflow documentation: https://www.tensorflow.org/api_docs
- Keras documentation: https://keras.io/api/
- Class materials

LINKS

- Dataset used: https://www.kaggle.com/datasets/tourist55/alzheimers-dataset-4-class-of-images