OCT DEMO
(keras - issue)

OBJECTIVE

The aim of this demo is to show the potential of image recognition in the medical field, using a dataset and deep learning techniques.
These methods could in the future assist the medical specialist by reducing cognitive effort and decreasing error, especially in ambiguous situations.
where the disease is in its infancy and therefore barely visible.

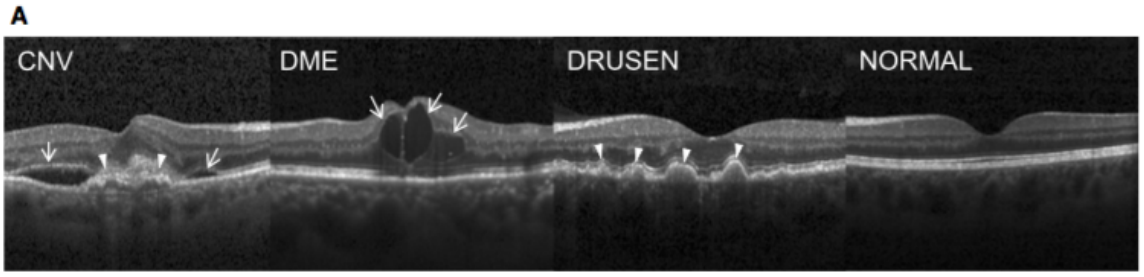
ISSUE BEFORE READ THE REPORT :

The main problem encountered, that keras is used for small datasets, is it has much lower performance than tensorflow or pyTorch.
This is why the training took too long, so for an implementation of a problem with real objects and images, it is not advisable to use the keras framework.
Here you can see the benchmarks between pyTorch and keras (https://wrosinski.github.io/deep-learning-speed-vol1/).
So it was decided to switch to pyTorch , to implement models with a framework also used in research and corporate.
keras has the characteristic of being easier to use and organise, but given its low performance, being a high-level framework, we were forced to change the framework.
That is why the explainability and the various tests of the written architectures were not continued.
The report below will describe the work done in the first moments, up to the interruption. The report and all the precautions taken with regard to the code have been scrupulously reviewed, trying to bring the keras framework to its full potential.
Work has not been interrupted, but implementations using pyTorch will appear in the near future.

DATASET

[Retinal OCT Images (optical coherence tomography) | Kaggle]
the dataset contains about 85k images, divided into training, test and validation set folders.
The classes within the dataset are 4: normal, drusen, CNV, DME

Example of Data for each class .



the images are in black and white

ARCHITECTURE USED :

COMPARISON :

Several tests were carried out, without taking the various architectures through to the end of the training, in order to assess performance prematurely, which is not a very correct thing to do, but we had to contend with the limitations of google colab, which limited resources and was too time-consuming at first to understand the best model.
We then considered the networks that performed best with the same epoch.

All data were done without performing augmentation and limiting the steph per epoch to 100, so 100*bat_size (64)

ARCHITECTURE	Accuracy	Loss	Epoch
AlexNet	62	1.33	1
Mnist Based	72	0.87	5
Custom *	-	-	-
Inception v3	74	0.74	2
EfficentNet	35	1.6	4

From this analysis we understand which networks to focus on.
*too much time to elaborate one epoch

Obviously, pre-trained networks, are faster in processing, in fact we started from the latter, between the two transfer learning techniques, we preferred to perform the feature reuse, because the classes of our problem, are very different from those used to train the network.

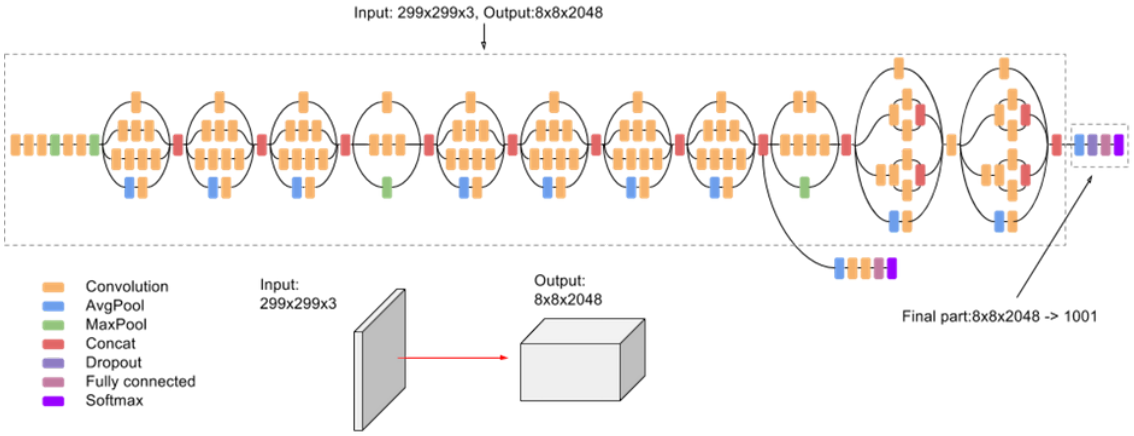
ALEXNET - from scratch

alexnet turned out to be optimal for this type of problem, we trained it from scratch.
Dropout layers were added and all the methods designed to decrease overfitting.
Tests were carried out both with and without augmentation, but as the dataset is very large, this may not be required, especially if we want to reduce the training time, which in the case of from scratch is already very high.
The training was interrupted for the change from keras to pyTorch, as the training took too long for the available resources.
This is the architecture of AlexNet :

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 48, 48, 32)        320

 batch_normalization (BatchN  (None, 48, 48, 32)       128
 ormalization)

 max_pooling2d (MaxPooling2D  (None, 24, 24, 32)       0
 )

 dropout (Dropout)           (None, 24, 24, 32)        0

 conv2d_1 (Conv2D)           (None, 22, 22, 64)        18496

 batch_normalization_1 (Batc  (None, 22, 22, 64)       256
 hNormalization)

 max_pooling2d_1 (MaxPooling  (None, 11, 11, 64)       0
 2D)

 dropout_1 (Dropout)         (None, 11, 11, 64)        0

 conv2d_2 (Conv2D)           (None, 9, 9, 128)         73856

 batch_normalization_2 (Batc  (None, 9, 9, 128)        512
 hNormalization)

 max_pooling2d_2 (MaxPooling  (None, 4, 4, 128)        0
 2D)

 dropout_2 (Dropout)         (None, 4, 4, 128)         0

 conv2d_3 (Conv2D)           (None, 2, 2, 256)         295168

 batch_normalization_3 (Batc  (None, 2, 2, 256)        1024
 hNormalization)

 max_pooling2d_3 (MaxPooling  (None, 1, 1, 256)        0
 2D)

 dropout_3 (Dropout)         (None, 1, 1, 256)         0

 flatten (Flatten)           (None, 256)               0

 dense (Dense)               (None, 512)               131584

 dense_1 (Dense)             (None, 512)               262656

 dense_2 (Dense)             (None, 4)                 2052

=================================================================
Total params: 786,052
Trainable params: 785,092
Non-trainable params: 960
_________________________________________________________________
```

INCEPTION V3 - pre-trained

By using inceptionv3 in this case using a transfer earning technique, i.e. fine tuning, an attempt was made to speed up the training process, while still achieving results of around 90% accuracy. Preliminary analyses on the first epochs showed promising results.
feature reuse was used as a transfer learning technique, because as inception is trained on totally different images, feature reuse is the most suitable.
In addition, the last layer for classification was eliminated and an external classifier was added, shown in the figure below immediately after the architecture :




```
[ ]  from tensorflow.keras.optimizers import RMSprop
     from tensorflow.keras import layers


     # Flatten the output layer to 1 dimension
     x = layers.Flatten()(pre_trained_model.output)
     # Add a fully connected layer with 1,024 hidden units and ReLU activation
     x = layers.Dense(1024, activation='relu')(x)
     # Add a dropout rate of 0.2
     x = layers.Dropout(0.2)(x)
     # Add a final sigmoid layer for classification
     x = layers.Dense  (4, activation='softmax')(x)
```

Small recap on the architecture and why it was chosen :

InceptionV3 is a deep convolutional neural network (CNN) architecture that was developed by Google researchers as part of the Inception family of models. Inception has been trained on ImageNet dataset that contains millions of images and more than 1000 classes.

Keras' implementation of InceptionV3 includes multiple layers of convolutional and pooling operations, as well as shortcut connections that allow information to bypass some of the layers. This architecture was designed to improve the accuracy of image classification models while keeping the number of parameters relatively low.

TECHNIQUE USED:

All these techniques were used to improve the models to the best of their ability. Various tests were carried out, with many hyper-parameters for each technique used.

Early stopping :

used to diminish the effect of overfitting, so training stops when the validation loss does not increase with a patience of 5 epochs.

BatchNormalization layer :

Overall, the BatchNormalization layer is a powerful tool for improving the performance and stability of deep learning models, and is widely used in state-of-the-art CNN architectures.

Normalizes the inputs to a layer by subtracting the batch mean and dividing by the batch standard deviation.

Augmentation :

in this case it was not used in the initial training models, because we had 85k images, but in the case of a real application it is to be used, because it succeeds in making the network generalise, minimising overfitting

Normalisation Layer :

tf.keras.layers.Rescaling(1./255)

it scales each input value by 1/255. This is often used for image data, where the pixel values are typically between 0 and 255, and scaling them to a range of 0 to 1 can improve model training.

Class weight :

It is used to handle with class imbalance dataset , as in this case.

It gives much more importance to the minority classes in the training phase, in order to decrease overfitting and improve the quality of the model, without undersampling which would decrease training.

was done using the tensorflow libraries, and for 85k data, it is done in less than 4 seconds.

Cache :

1. prefetch(buffer_size=AUTOTUNE) applies background prefetching to the data, which means that the data will be loaded in the background while the model is training on the previous batch. This can help to overlap the time spent on I/O operations
2. cache() applies in-memory caching to the data, which means that the data will be loaded from disk and stored in memory on the first epoch and then read from the cache on subsequent epochs. This can speed up training by reducing the time spent on disk I/O operations.

Dropout Layer :

Regularization technique that is used to reduce overfitting.

In our case Dropout layer with a rate of 0.2 is added to the model after the Flatten layer. This means that 20% of the inputs to the next layer will be randomly set to zero during each training step.

Research has shown that several small dropouts are much better than a single 0.5 dropout at the end, so this policy was applied

IMPROVEMENT :

- Use all the augmented data to train the model
- Hyperparameter tuning with all parameter
- **pyTorch SWITCH**