CT Scan Classification

Ana-Maria Comorașu

May 2021

1 Introduction

1.1 Purpose of the project

The purpose of the project is to train a model for a Computer Tomography (CT) scan classification. This is a multi-way classification task in which an image must be classified into one of the three classes (native, arterial, venous).

1.2 Input Data

- training data - 15,000 images
- validation data - 4,5000 images
- test data - 3,900 images

For implementation purposes, the training and validation images are separated in 3 directories, depending on what class they belong to.

1.3 Data Augmentation

```
training_data = ImageDataGenerator(
    rotation_range=20.,
    width_shift_range=0.2,
    height_shift_range=0.2,
    zoom_range=0.2,
    data_format='channels_last'
)
```

1.4 Machine Learning Approaches

All tried models are based on deep learning approach, which is Convolutional Neural Network (CNN). This approach is widely used in solving complex problems and is most commonly applied to analyze visual imagery. For an image classification task, CNN seemd like the most reliable approach, compared to KNN, Naive Baise or SVM.

Taking into account this decision, there are two ways in which I tried implementing the required model:

1. By implementing a sequential CNN architecture
2. By using a pre-trained CNN architecture

1.5 Python Libraries

- Keras API and Tensorflow
- Sklearn, Matplotlib, Numpy
- Pandas, Os, Zipfile

2 Sequential Architecture CNN

2.1 CNN Introduction

A Convolutional Neural Network (CNN) is a Deep Learning algorithm which can take an image as an input, assign importance to its various aspects/objects and be able to differentiate one from other. The architecture of a CNN is analogous to that of the connectivity pattern of neurons in the human brain so that individual neurons respond to stimuli only in a restricted region of the visual field known as the receptive field.

2.2 Search and Tryouts

2.2.1 Convolutional Layers

A convolutional layer is the first layer to extract high-level features from an input image. It preserves the relationship between pixels by learning image features using small squares of input data.

For each convolutional layer, the activation ReLU is added - which stands for Rectified Linear Unit for a non-linear operation.

The pooling layers would reduce the number of parameters when the images are too large. Max Pooling takes the largest element from the rectified feature map. As a first experiment, there were implemented 3 basic models:

- One convolutional layer with 24 filters, with kernel size = 5, relu activation and a MaxPooling2D layer: $[24C5 - P2]$.
- The same as precedent, adding a convolutional layer with 48 filters and the same parameters as before and MaxPooling2D layer: $[24C5 - P2] - [48C5 - P2]$
- Same as before, plus a convolutional layer with 64 filters and a MaxPooling2D layer: $[24C5 - P2] - [48C5 - P2] - [64C5 - P2]$

For all implementation descriptions above, there were added the following layers:

- Flatten layer
- A dense layer with 256 units and relu activation.
- A dense layer with 3 units and softmax activation (final layer).

All models were compiled with Adam optimizer for 20 epochs.

The results are, with epochs = 20:

Model	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
1 Conv Layer	3.7975e-04	1.0000	6.7605	0.5311
2 Conv Layers	0.0014	0.9999	5.5750	0.5962
3 Conv Layers	0.0011	0.9997	3.7613	0.6971

Chosen version: 3 convolutional layers

2.2.2 Number of filters in the Convolutional Layer

The element involved in carrying out the convolutional operation is the kernel/filter. The search was between the following models:

- $[8C5 - P2] - [16C5 - P2] - [32C5 - P2] - F - D256 - D3$
- $[16C5 - P2] - [32C5 - P2] - [64C5 - P2] - F - D256 - D3$
- $[24C5 - P2] - [48C5 - P2] - [96C5 - P2] - F - D256 - D3$
- $[32C5 - P2] - [64C5 - P2] - [128C5 - P2] - F - D256 - D3$

- $[48C5-P2]-[96C5-P2]-[160C5-P2]-F-D256-D3$
- $[64C5-P2]-[128C5-P2]-[192C5-P2]-F-D256-D3$

The results are, with epochs = 20:

Model	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
[8C5-P2]	0.0435	0.9858	2.3208	0.6967
[16C5-P2]	0.0414	0.9865	2.2930	0.6944
[24C5-P2]	0.0459	0.9824	1.9070	0.7176
[32C5-P2]	0.0162	0.9940	2.3304	0.7022
[48C5-P2]	0.0237	0.9922	1.8146	0.7278
[64C5-P2]	0.0267	0.9898	1.9296	0.7040

Chosen version: $[48C5-P2]-[96C5-P2]-[160C5-P2]-F-D256-D3$

2.2.3 Dense layer units

The layer called fully connected layer, is flattening the matrix into a vector end fed into a fully connected layer like a neural network. The dense layer units was chosen between the following models:

- $[48C5-P2]-[96C5-P2]-[160C5-P2]-F-D0-D3$
- $[48C5-P2]-[96C5-P2]-[160C5-P2]-F-D32-D3$
- $[48C5-P2]-[96C5-P2]-[160C5-P2]-F-D64-D3$
- $[48C5-P2]-[96C5-P2]-[160C5-P2]-F-D128-D3$
- $[48C5-P2]-[96C5-P2]-[160C5-P2]-F-D256-D3$
- $[48C5-P2]-[96C5-P2]-[160C5-P2]-F-D512-D3$
- $[48C5-P2]-[96C5-P2]-[160C5-P2]-F-D1024-D3$
- $[48C5-P2]-[96C5-P2]-[160C5-P2]-F-D2048-D3$

The results are, with epochs = 20:

Model	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
D0	0.0156	0.9954	2.3936	0.6967
D32	0.0045	0.9991	2.2577	0.7147
D64	0.0012	0.9999	2.3416	0.7278
D128	0.0017	0.9998	2.7592	0.7140
D256	0.0029	0.9995	2.3885	0.7353
D512	0.0103	0.9973	2.4650	0.6956
D1024	9.7777e-04	0.9998	2.3874	0.7282
D2048	0.0020	0.9993	3.3198	0.7107

Chosen version: $[48C5-P2]-[96C5-P2]-[160C5-P2]-F-D64-D3$

2.2.4 Dropout

Large CNNs trained on a relative small dataset can overfit the training data, which can result to poor performance when the model is evaluated on a new dataset. Dropout is a regularization method that approximates training a large number of neural networks with different architectures in parallel. The dropout value was chosen between the following:

- $[48C5-P2]-[96C5-P2]-[160C5-P2]-F-D64-Drop0.0-D3$
- $[48C5-P2]-[96C5-P2]-[160C5-P2]-F-D64-Drop0.1-D3$

- $[48C5 - P2] - [96C5 - P2] - [160C5 - P2] - F - D64 - Drop0.2 - D3$
- $[48C5 - P2] - [96C5 - P2] - [160C5 - P2] - F - D64 - Drop0.3 - D3$
- $[48C5 - P2] - [96C5 - P2] - [160C5 - P2] - F - D64 - Drop0.4 - D3$
- $[48C5 - P2] - [96C5 - P2] - [160C5 - P2] - F - D64 - Drop0.5 - D3$
- $[48C5 - P2] - [96C5 - P2] - [160C5 - P2] - F - D64 - Drop0.6 - D3$
- $[48C5 - P2] - [96C5 - P2] - [160C5 - P2] - F - D64 - Drop0.7 - D3$

The results are the following, with epochs = 15:

Model	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
Drop 0.0	0.0661	0.9762	1.6120	0.7207
Drop 0.1	0.1109	0.9569	1.0856	0.7349
Drop 0.2	0.1655	0.9322	1.0414	0.7387
Drop 0.3	0.2526	0.8967	0.7549	0.7380
Drop 0.4	0.3350	0.8543	0.8244	0.7291
Drop 0.5	0.4277	0.8012	1.5185	0.6613
Drop 0.6	0.5702	0.7248	1.3541	0.6191
Drop 0.7	0.7560	0.6334	1.0302	0.5753

Chosen version: $[48C5 - P2] - [96C5 - P2] - [160C5 - P2] - F - D256 - Drop0.2 - D3$

2.3 Best implementations found

Batch Normalization is a technique for training neural networks that standardizes the input to a layer for each mini-batch. This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train CNNs.
After testing, searching, adding Batch Normalization and more Dropout, a good model was the one below:

```
model = Sequential()
model.add(BatchNormalization(input_shape=(50, 50, 1)))

model.add(Conv2D(64, kernel_size=5, activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(128, kernel_size=3, activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(BatchNormalization())
model.add(Dropout(0.1))

model.add(Conv2D(256, kernel_size=3, activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.2))

model.add(Conv2D(512, kernel_size=3, activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=2))

model.add(Flatten())
model.add(Dense(1024, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(3, activation='softmax'))
model.compile(optimizer=Adam(),
    loss=tf.keras.losses.categorical_crossentropy,
    metrics=['accuracy'])
```

This implementation was tested with 3 different number of epochs: 25, 100 and 250. The last one had the best accuracy

Model	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
250 epochs	0.0204	0.9941	2.8167	0.8151

2.4 Confusion matrix for the best option

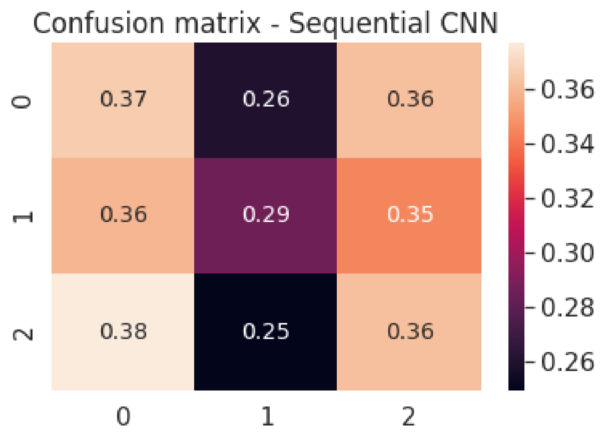


Table 1: Classification Report - Sequential CNN Model

Class	Precision	Recall	F-score	Support
0	0.33	0.37	0.35	1500
1	0.36	0.29	0.32	1500
2	0.34	0.37	0.35	1500
accuracy			0.34	4500
macro avg	0.34	0.34	0.34	4500
weighted avg	0.34	0.34	0.34	4500

3 Pretrained CNN Architectures

Among many well known and new methods which contribute to deep learning, transfer learning can be found. This is a method which uses the representations learned by one trained model for another model that needs to be trained on different data for another task. Transfer learning uses pre-trained models, already trained on larger datasets like Imagenet.

3.1 ResNet50

ResNet50 is a variant of ResNet model, being made up of 48 Convolutional layers, 1 MaxPooling layer and 1 AveragePooling layer. A residual network distinguishes itself from other CNNs is because it uses identity connection between layers.

3.2 Code example

```
model = ResNet50(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

def layer_adder(bottom_model):
  top_model = bottom_model.output
  top_model = GlobalAveragePooling2D()(top_model)
```

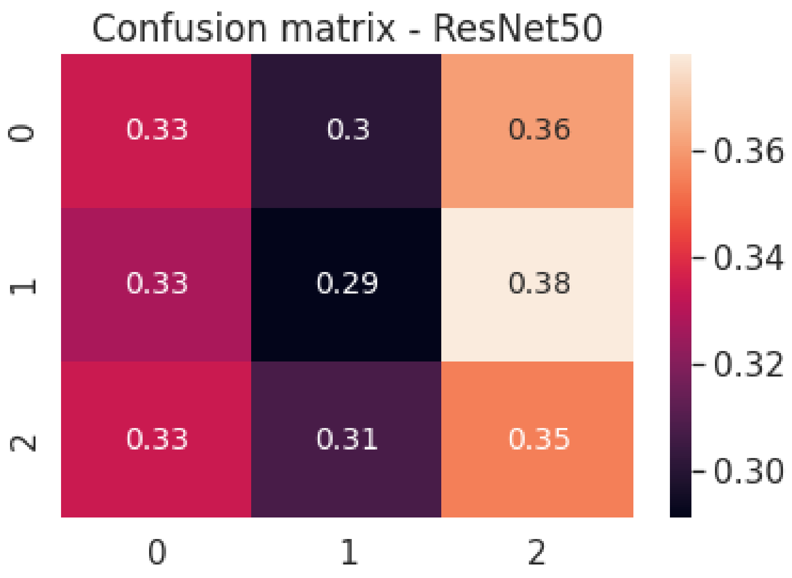
```
    top_model = Dense(1024,activation='relu')(top_model)
    top_model = Dense(3,activation='softmax')(top_model)
    return top_model

head = layer_adder(model)
resnet_model = Model(inputs=model.input, outputs=head)
```

3.3 Classification report and Confusion matrix

Model	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
ResNet50 - 30 epochs	0.0021	0.9993	0.5685	0.8998

Table 2: Classification Report - ResNet50

Class	Precision	Recall	F-score	Support
0	0.33	0.33	0.33	1500
1	0.33	0.30	0.31	1500
2	0.32	0.35	0.34	1500
accuracy			0.33	4500
macro avg	0.33	0.33	0.33	4500
weighted avg	0.33	0.33	0.33	4500



4 Conclusions

A pretrained model has many more benefits than a basic model with 3 convolutional layers, even though it is tuned. As observed in the models above, using a pretrained model improved with about 10% in accuracy.