

Chapter 3 – Realization of an algorithm for the identification and segmentation of brain tumors from images, by using convolutional neural networks

This algorithm aims to be able to identify those MRI images that have brain tumor and identify the type of abnormality, and the goal is to achieve greater than 90% accuracy on the test set. It will be built in Python 3.6, and to build it, I've chosen to use a dataset split into a training set and a test set. With the help of this data set, the created model was trained and thus, it can identify a brain tumor and classify it into *glioma*, *meningioma*, *pituitary tumor* or to specify whether the patient to whom the MRI image corresponds has no tumor.

The diagram of the algorithm can be seen in the figure below:

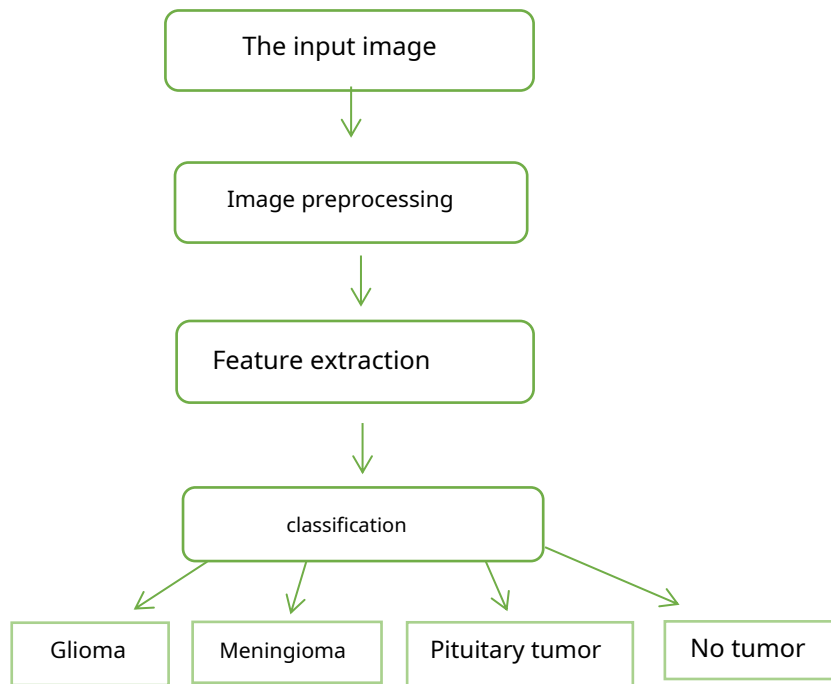


Figure 3.1. – Scheme of the proposed algorithm for identifying and classifying brain tumors

3.1. Data set preparation

In this kind of algorithms, the data set is extremely important because it is the basis for the correct functioning of the network. That is why it is essential that the images used in the creation of the network

convolutional neural networks, to be in as large a number as possible and in the optimal form for obtaining the best results and predictions.

For this work, the database used is taken from the Kaggle website and consists of specific magnetic resonance images (MRI) of the brain. Unfortunately, the provenance of these images or the date this database was created is unknown.

Each set is further divided into four categories that label the types of brain tumors they represent: glioma, meningioma, pituitary, and nontumor. So, the database includes 5712 images in the training set, of which for the "glioma" category, there are 1330 pictures, for "meningioma" 1348, for pituitary 1430 and for the "no tumor" category, 1604 images. The test set is considerably smaller, consisting of 1311 pictures, of which there are 300 images for glioma, 405 for meningioma, 300 for pituitary, and 306 without tumor.

For a better organization of them, I created in Google Drive a folder called "Dissertation data", in which I created two other folders: Training and Testing, which in turn contain four other folders specific to each of the previously mentioned categories. It should be noted that I was not the one who determined the category to which each picture belongs, but this aspect was already established by authorized persons in this sense.

After uploading the data to create the model, I want to view some images to make sure that the upload was successful. Thus, in the image below you can see an example from each category in the training set.

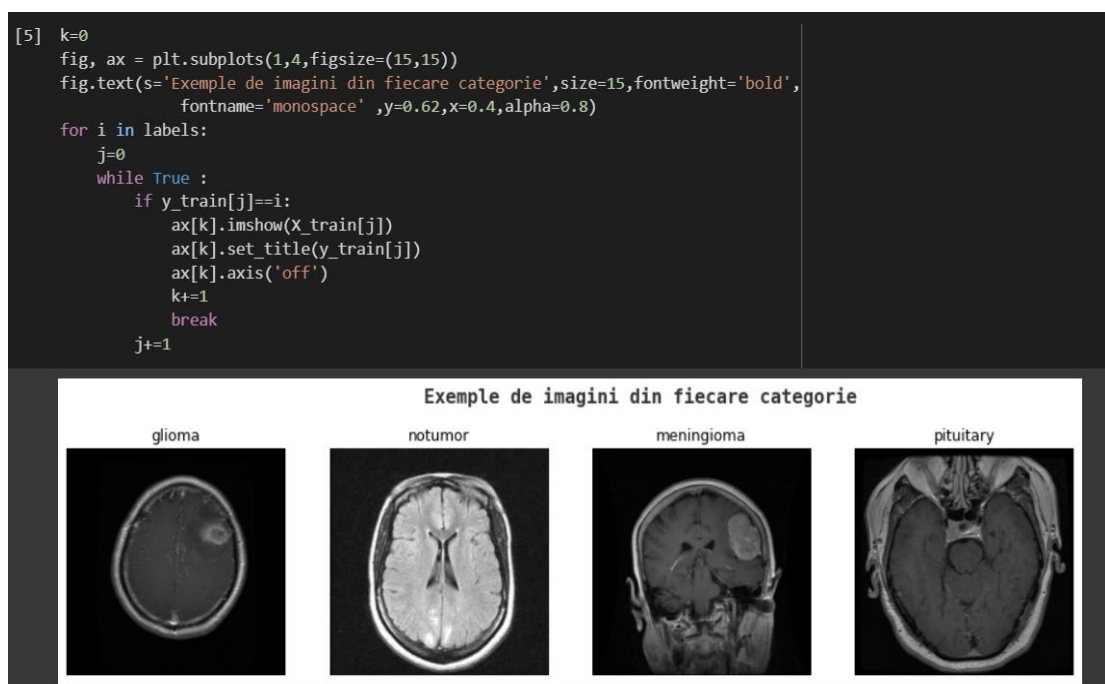


Figure 3.2. – Example images from each category in the training set.

- Data preprocessing

Preprocessing refers to all the transformations performed on the raw data before it is fed into the machine learning or deep learning algorithm. For example, training a convolutional neural network based on raw images will likely result in poor classification performance. [73]

Thus, to prepare the set of images for training, validation and testing, a resizing operation was used to resize the entire data set to 224 x 224. This size was used to match the input required by the network we are going to build. This step is important because it helps the convolutional network to work better, since the images are not the same size and thus it is harder to find the specific features.

```
[ ] X_train = []
    y_train = []
    X_test=[]
    y_test=[]
    image_size = 224
    for i in labels:
        folderPath = os.path.join('/content/gdrive/My Drive/Date disertatie','Training',i)
        for j in tqdm(os.listdir(folderPath)):
            img = cv2.imread(os.path.join(folderPath,j))
            img = cv2.resize(img,(image_size, image_size))
            X_train.append(img)
            y_train.append(i)

    for i in labels:
        folderPath = os.path.join('/content/gdrive/My Drive/Date disertatie','Testing',i)
        for j in tqdm(os.listdir(folderPath)):
            img = cv2.imread(os.path.join(folderPath,j))
            img = cv2.resize(img,(image_size,image_size))
            X_test.append(img)
            y_test.append(i)

    X_train = np.array(X_train)
    X_test= np.array(X_test)
```

Figure 3.3 – The process of uploading and resizing images to 224 x 224

```
[ ] X_train.shape

(5712, 224, 224, 3)
```

Figure 3.4 – The new shape of the training set

So, in addition to the resizing shown in Figure 3.3, we start by adding all the images in the directories to a Python list to convert them to arrays *numpy*. In addition, the code sequence above also adds the corresponding labels to the images represented by *y_train* and *y_test*.

- Data set shuffling

The next step in data set preparation is to shuffle the data set. The purpose of this step is to reduce the variance and ensure that the model remains as general as possible, so as to avoid overfitting, i.e. when the accuracy for the training data is high, but for the test set, it is not. Generally, this step is done when the data set is sorted by category, as it happens in the present case, where we have four different categories, the images being in order for each category. In the constructed algorithm, we have achieved this using the method *shuffle* which is intended to change the order of the elements in a list without returning a new one instead.

```
[ ] x_train, y_train = shuffle(x_train, y_train, random_state=28)
```

Figure 3.5 – Shuffle dataset by function *shuffle*

After doing this process, we can visualize the distribution of images on each category separately, for both the training set and the test set.

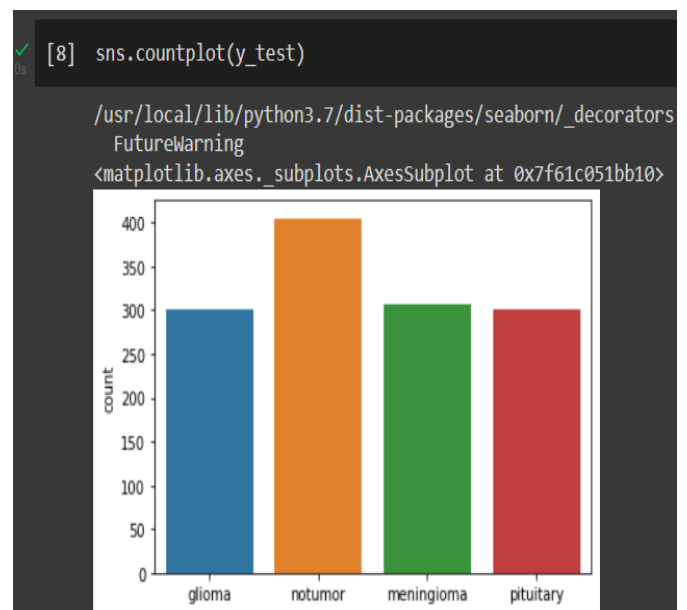
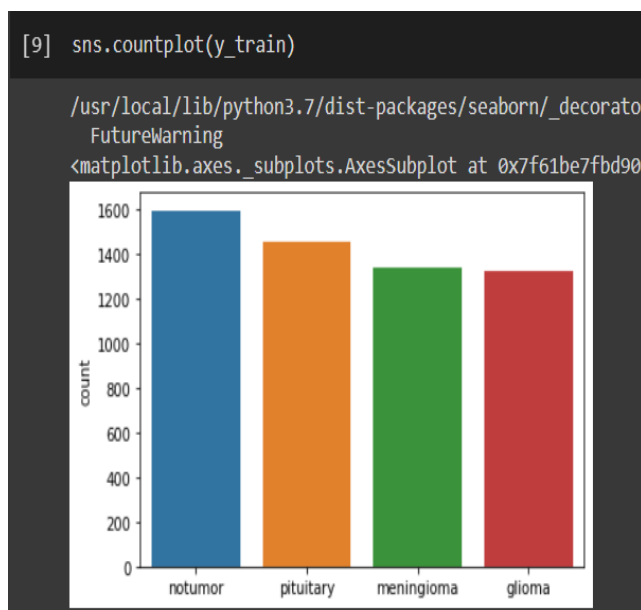


Figure 3.6 – Distribution of data according to the category they belong to
(Left – training set, right – test set)

- One Hot Encoding

After the data set is scrambled, a step known in Machine Learning algorithms as "One-Hot Encoding" follows. We know that when we train a neural network through supervised learning, the model predicts an outcome. In the case of image classifiers, the network receives as input images that have a certain label. In the present case, the images are labeled according to the brain tumor they represent. But in these situations, the created model does not know how to interpret those words like "glioma", "meningioma", "pituitary" or "notumor", nor to provide them with outputs in the form of labels. So, most of the time, these names are encoded to take the form of an integer or a vector of integers. For this, a type of encoding used for ML algorithms,

Glioma - [x, x, x, x] - [1, 0, 0, 0] - 1

No Tumor - [x, x, x, x] - [0, 1, 0, 0] - 2

Meningioma - [x, x, x, x] - [0, 0, 1, 0] - 3

Pituitary - [x, x, x, x] - [0, 0, 0, 1] - 4

Each element in the vector corresponds to a category, so we can assume that "Glioma" corresponds to the first element in the vector, "No Tumor" to the second, "Meningioma" to the third, and "Pituitary" to the fourth. It can be seen from the diagram above how each category has a certain position in the corresponding vector, marked by the digit 1. Thus, as the name of the method suggests, it can be deduced that each created vector consists of digits of 0, except element of the position corresponding to each category. Therefore, through this method, the created model can "read" the images and the categories they belong to, and the vectors corresponding to each one will look like those in the diagram above.

```
[ ] y_train_ = []
    for i in y_train:
        y_train_.append(labels.index(i))
    Y_train = y_train_

    Y_train = tf.keras.utils.to_categorical(Y_train)

    y_test_ = []
    for i in y_test:
        y_test_.append(labels.index(i))
    Y_test = y_test_

    Y_test = tf.keras.utils.to_categorical(Y_test)
```

Figure 3.7 – One-Hot Encoding

In the figure above, it is shown how the encoding of the labels was done for both the training set and the test set. The command used is *tf.keras.utils.to_categorical*, and this converts a vector of classes to an array of binary classes.

- Splitting the data set

In the process of making a neural network, it is necessary to divide the data set into three subsets: *the training set*, *the validation set* and *the test set*.

The training set—is that subset that is used to train and make the model learn all the features of the images given as input. Every *epoch* (Epoch is a hyperparameter that defines how many times the algorithm traverses the training data set. A single epoch represents that each image in the training set has had a chance to update the model's parameters), the network architecture is repeatedly fed the same training data, and the model continuously learns all of its features. It is advisable that this subset of data be as diverse as possible to allow the model to encompass all scenarios and ultimately be able to predict any unseen data sample. The training set must be the largest compared to the other two subsets [74].

The validation set—This is a dataset used to validate the model's performance during training. The data validation process returns information that is helpful in tuning hyperparameters and model configurations.

In other words, I am like a critic who says if the model is going in the right direction. Validation of the model on the validation set is performed after each epoch.

The main reason for splitting the dataset into validation and training is to avoid overfitting the model. That is, one does not want the model to become very good at classifying data from the input set, and very poor at classifying new data, which it has not seen before [74].

The test set–This is a completely separate set of data used to test the performance of the model after it has already been trained. This is the set that helps us answer the final question after the model is built: "How well does it actually work?".

```
[ ] X_train, X_val, Y_train, Y_val = train_test_split(X_train, Y_train, test_size=0.2, random_state=28)
```

Figure 3.8 – Splitting the training set into training subset and validation subset

In figure 3.8, it can be seen how the data was loaded and how the test set and the training set are formed. The test set remains intact until the process is complete, but the training set is split into the other two mentioned subsets. In the code shown in the figure above, this is done with the help of the function *train_test_split()*, made available through the bookstore *sklearn*. So, from the 5712 images that were initially in the training set, 20% are used by the validation set.

3.2. Building the model

In the following, I will describe the steps taken to build the convolutional neural network model, explaining the operation of certain concepts used in this process.

- Transfer Learning – creation of the convolutional neural network model

Training many convolutional neural network models can take hours or even days for very large datasets. To shorten this process, a procedure very often

used in Machine Learning algorithms is transfer learning, where a model already developed for a specific purpose is reused, as a starting point for the development of a new model that fulfills the purpose desired by the developer. This is a very popular approach in deep learning because these existing models are based on very powerful computing resources and huge datasets.

In developing the brain tumor classification algorithm, we used the model *EfficientNetB0* which uses the weights from the ImageNet dataset.

- EfficientNetB0

EfficientNet was first introduced in 2019 by Google researchers and is considered one of the most efficient models that achieves very good accuracy for image classification tasks. This is a convolutional neural network architecture and method for scaling all dimensions of depth, width and resolution using a composite coefficient for this. In general, classical algorithms for convolutional neural networks are based on scaling only the depth, i.e. the number of layers used for image classification. The researchers were able to find the appropriate scaling factors for each of the dimensions by *compound scaling method* which uses a compound coefficient α to scale width, depth and resolution together. So instead of randomly increasing the three dimensions, compound scaling increases uniformly on each of them. Using the scaling method and AutoML, the authors developed seven EfficientNet models of different sizes (from B0 to B7), surpassing the state-of-the-art accuracy of most convolutional neural networks [75, 76].

- Depth – the number of layers. Refers to how deep the network is.
- Width – the number of channels in a convolution layer. In other words, how wide the net is.
- Resolution – refers to the resolution of the images used as input.


```
[ ] base_model = EfficientNetB0(weights='imagenet', include_top=False, input_shape=(image_size, image_size, 3))

Downloading data from https://storage.googleapis.com/keras-applications/efficientnetb0\_notop.h5
16711680/16705208 [=====] - 0s 0us/step
16719872/16705208 [=====] - 0s 0us/step

[ ] model = base_model.output
model = tf.keras.layers.GlobalAveragePooling2D()(model)
model = tf.keras.layers.Dropout(rate=0.5)(model)
model = tf.keras.layers.Dense(4,activation='softmax')(model)
model = tf.keras.models.Model(inputs=base_model.input, outputs = model)
```

Figure 3.9 – Downloading the EfficientNetB0 model and implementing it

The first line of code in Figure 3.9 represents the download of the model I will use, namely EfficientNetB0. parameter *include_top* is set to "False" because we do not want the network to include the output layer from the pre-built model. This allows you to create your own output layer to match your chosen dataset. This template requires images to be 224 x 224 in size.

After that, create the other layers in the following order:

- a **GlobalAveragePooling2D**(tf.keras.layer.GlobalAveragePooling2D) – this is the convolutional neural network pooling layer and uses the Average Pooling method, meaning it will use average values when pooling, instead of using the maximum value as with Max Pooling. I chose to use Average Pooling because, in addition to improving the computational load during training, it reduces image noise better and captures all features in the pooling region, compared to Max Pooling which may ignore some background features. Due to this aspect, Average Pooling helps better in identifying anomalies in medical images [77].
- a **Dropout**(tf.keras.layer.Dropout) – This layer is used to "turn off" part of the neurons at each step in the layer so that they are more independent from neighboring neurons. This helps to avoid overfitting. The neurons to be omitted are chosen at random, and the parameter *installments* represents the probability that a neuron is ignored. In general, the most often used value is 0.5, so I chose that value as well.

- a **Dense**(tf.keras.layers.Dense) – This command builds the fully connected layer, which classifies the image into one of four classes. This layer uses the softmax activation function (activation='softmax'). This activation function is usually used in output layers because it helps to convert number vectors into probability vectors. Each value is the probability of belonging to each class.
 - a **Model**(tf.keras.models.Model) – It is a standard command used in creating convolutional neural network models, being the last step in this process. The command groups layers into a single feature object.
- Compiling the created model

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

Figure 3.10 – Compiling the model

Finally, we arrived at the compilation of the model, which is done only by the method presented in Figure 3.10, *compile*. In general, this has a standard form that requires three parameters: the parameter that calculates the loss (loss) being of the type "categorical_crossentropy", the optimizer (optimizer) with the value "Adam" which is a function that aims to minimize the loss of the model and improve the accuracy. The third parameter, metrics is set with the value "accuracy" and helps to measure the accuracy of the model.

- Defining callbacks

This step is important in building Deep Learning algorithms as it helps to resolve errors and improve models. During training, the values defined in the code in Figure 3.10, allow following the evolution of the model created during training. It also helps prevent overfitting by implementing "Early Stopping" or changing the learning rate at each iteration.

```
reduce_lr = ReduceLROnPlateau(monitor='val_accuracy', factor=0.3, patience=2, min_delta=0.0001, mode='auto', verbose=1)
tensorboard = TensorBoard(log_dir='logs')
checkpoint = ModelCheckpoint("brain_tumor_classification.h5", monitor='val_accuracy', save_best_only=True, mode='auto', verbose=1)
```

Figure 3.11 – Defining callbacks

In the developed algorithm, three functions are used:

- a **ReduceLROnPlateau**–The written function reduces the learning rate when the accuracy of the model no longer improves from one epoch to another, an aspect marked by the parameter `monitor='val_accuracy'`. parameter `factor=0.3`, represents the factor by which the learning rate is reduced, and the parameter `patience=2` is the number of epochs where the accuracy does not improve that the model waits before reducing the learning rate. parameter `min_delta` is the new optimal measurement threshold to focus only on significant changes, `mode='auto'` helps to reduce the learning rate when the accuracy is either too high or too low, and `verbose=1` it simply provides an update message as seen in the figure below.

```
143/143 [=====] - ETA: 0s - loss: 0.0032 - accuracy: 0.9991
Epoch 33: val_accuracy did not improve from 0.98513

Epoch 33: ReduceLROnPlateau reducing learning rate to 1.5943230069481729e-10.
143/143 [=====] - 65s 455ms/step - loss: 0.0032 - accuracy: 0.9991 - val_loss: 0.0775 - val_accuracy: 0.9843 - lr: 5.3144e-10
Epoch 34/50
```

Figure 3.12 –ReduceLROnPlateau update message when the learning rate is reduced during training

- a **TensorBoard** –It is very helpful that while learning, you can measure the evolution of the model at each step. The TensorBoard feature helps track metrics like accuracy and loss.
- a **ModelCheckpoint** –This function is used to save the weights of the model that is considered the best from the point of view of the monitored value, the accuracy in this case, marked by `save_best_only=True`.

- Data augmentation

The next step in creating the algorithm is to augment the data set. Since deep learning algorithms require very large datasets, the training dataset we used (5712 images) cannot be considered sufficient to achieve the best results. So this step is to increase the data set, because the more data you "feed" the neural network, the better results you get.

```
[31] datagen = ImageDataGenerator(  
    featurewise_center=False,  
    samplewise_center=False,  
    featurewise_std_normalization=False,  
    samplewise_std_normalization=False,  
    zca_whitening=False,  
    rotation_range=90,  
    zoom_range=0.2,  
    width_shift_range=0.2,  
    horizontal_flip=False,  
    vertical_flip=False)  
  
    datagen.fit(X_train)
```

Figure 3.13 – Data set augmentation by the ImageDataGenerator function

In the figure above, the code how the augmentation of the data set was achieved for this algorithm is detailed. The function was used *ImageDataGenerator* from the package *TensorFlow* available in the Python language that creates new batches of data in real time. In the present case, part of the process of enlarging the data set is done simply by rotating the images by 90 degrees with the parameter `rotation_range=90` degrees, or by zooming the images closer or further away by a certain value with the parameter `zoom_range=0.2` and by shifting the pictures horizontally with `width_shift_range=0.2`. Also, `featurewise_center=False` sets the input mean to 0 for the entire data set, `samplewise_center=False` sets the mean of each sample to 0, `featurewise_std_normalization=False` divides the input data by the standard deviation of the data set, `samplewise_std_normalization=False` does the same for samples, `zca_whitening` reduce size,

```
[17] def visualize(original, augmented):
    fig = plt.figure()
    plt.subplot(1,2,1)
    plt.title('Original')
    plt.imshow(original)

    plt.subplot(1,2,2)
    plt.title('Augmentat')
    plt.imshow(augmented)
    rotated=tf.image.rot90(X_train[1])
    visualize(X_train[1], rotated)
```

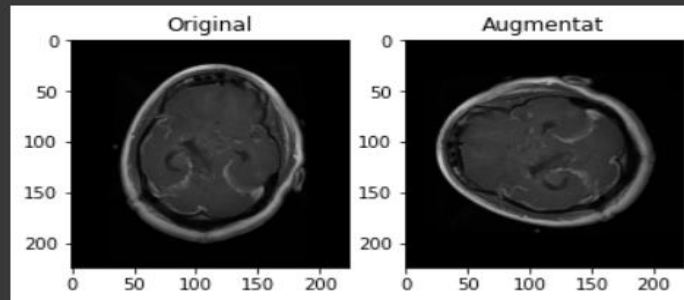


Figure 3.14 – Example image from the augmented data set, by rotating the image 90 degrees

- Training the model

```
history = model.fit_generator(datagen.flow(X_train, Y_train, batch_size=32),
                             validation_data=(X_val, Y_val),
                             epochs=50,
                             verbose=1,
                             callbacks=[tensorboard, checkpoint, reduce_lr])
```

Figure 3.15 – Convolutional neural network training

Next is the training of the network, when it learns the features of the images so that it is able to perform their classification. Training is done by function *model.fit_generator* which works very well in the case of supervised learning, as is the case here. Within it, the number of epochs is also determined. I chose their number to be 50 (epochs=50), because if the number of learning stages is much too high, the problem of overfitting may arise, therefore, I considered this number to be adequate.

- Viewing model performances

To check whether the created model overfits or behaves as expected, we can visualize the evolution of model accuracy and loss at each step.

For better clarity on this, I will break down what each of the two performance indicators mean. Accuracy is used to measure the accuracy of the model created in an interpretable way. This is generally determined by model parameters and is calculated as a percentage. In other words, it is that measure that provides information on how accurate the prediction of the created model is compared to the actual data. The loss function, on the other hand, is used to optimize algorithms. This is calculated on training and validation, and as an interpretation, it shows how well the model behaves on the two data sets: training and validation. It is a summation of all the errors made for each image in the two sets,

```
# Grafic pentru a vedea evolutia acuratetii și a pierderii in timp
fig, ax = plt.subplots(2,1)
ax[0].plot(history.history['loss'], color='b', label="Training loss")
ax[0].plot(history.history['val_loss'], color='r', label="validation loss", axes = ax[0])
legend = ax[0].legend(loc='best', shadow=True)

ax[1].plot(history.history['accuracy'], color='b', label="Training accuracy")
ax[1].plot(history.history['val_accuracy'], color='r', label="Validation accuracy")
legend = ax[1].legend(loc='best', shadow=True)
```

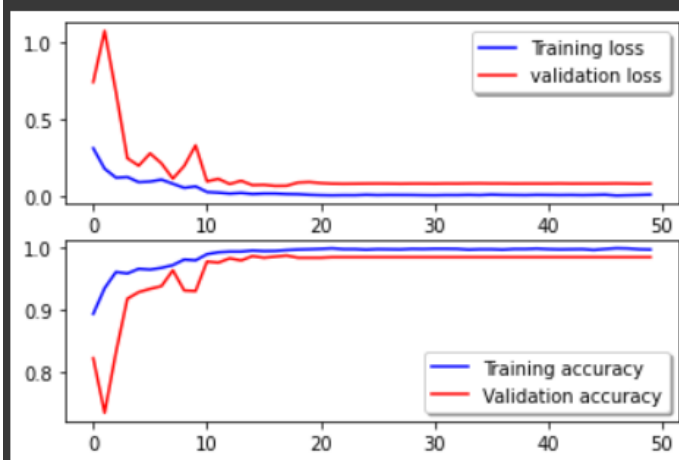


Figure 3.16 – Evolution of the loss function (top graph) and accuracy (bottom graph) in each epoch, for the training and validation set

In the first graph in Figure 3.14 it can be seen that the model has a fairly small and constant loss starting at epoch 10 for the training set (blue). As for the loss for the validation set (red), it can be seen how initially it has very high values, then starting from epoch 10, it is almost the same value as the one in

the training set. This indicates that the model starts with iteration 10 to perform as expected.

The same thing happens with accuracy, shown in the second graph. For the training set, the accuracy has an increasing evolution initially, then becomes constant from the 10th iteration, having high values. In contrast, for the validation set, from iteration 0 to about the 10th iteration, the model's accuracy value fluctuates being very small at first, followed by a very large increase. With epoch 10, as in the other cases, the model stabilizes and the accuracy for the validation set becomes constant, similar to that of the training set.

3.3. Evaluation of the model

At the end of any machine learning algorithm, the model created must be evaluated to ensure that it works correctly and does not produce overfitting. That is, in the evaluation phase, one wants to test the predictions on the test set we created previously, to see if it can correctly classify images it has never seen before. This is a very revealing step in the evaluation of the created model, because if it is not able to classify the unseen images with good accuracy, the obtained results are in vain for the training set.

```
[ ] pred = model.predict(X_test)
    pred = np.argmax(pred,axis=1)
    y_test_new = np.argmax(Y_test,axis=1)
```

Figure 3.17 – Prediction of the created model

So, as can be seen in Figure 3.15, the variable is created *teacher* with the help function *pred()* which uses as the only parameter the data set on which we want to make the prediction. In the present case, this dataset is *X_test*, created in point 3.1. The function is also used *np.argmax()* in the last line of code in the figure shown, because this Numpy function is used to find the argument that returns the maximum value from a target function. In other words, it is most often used in the prediction of multi-class classification models because they use probability vectors that show the probability as a

sample to belong to one of the created tags. (Y_test in the present case). So the argmax function converts this vector of probabilities into a class-specific label, finding the one with the highest predicted probability.

```
print(classification_report(y_test_new,pred))
```

	precision	recall	f1-score	support
0	1.00	0.99	0.99	300
1	1.00	1.00	1.00	405
2	0.99	0.97	0.98	306
3	0.96	1.00	0.98	300
accuracy			0.99	1311
macro avg	0.99	0.99	0.99	1311
weighted avg	0.99	0.99	0.99	1311

Figure 3.18 – The classification report

Another method to evaluate the performance of the created model is to check the classification report made available in the Python language by the function `classification_report()` from the sklearn bookstore. In the figure above, four indicators can be observed: precision, recall, f1-score and support, each with a unique interpretation.

It should be noted that each of the four figures belongs to a category, as can be seen below:

- 0- Glioma
- 1- No tumor
- 2- Meningioma
- 3- Pituitary tumor.

Precision—it is like a measure of quality, being considered the accuracy with which the model makes positive predictions. The higher its value, the more it means that the algorithm returns relevant results in most cases. From the displayed image, it can be seen that the algorithm has a very good ability to classify correctly, having values above 0.9. The best can classify images indicating glioma and no tumor (1.00), and the worst can classify pituitary tumors (0.96)

Recall—is the report that indicates the number of positive cases, correctly identified. It is calculated as a fraction of the number of positively identified cases and the sum of true positives

and false negatives. And for this identifier, a value as close as possible to 1 indicates a better network operation. And for this it is observed that the images from the category "No tumor" and "Pituitary tumor" are perfectly identified (1.00), and the lowest value indicates it for the other two categories (0.98).

F1-score–The F1 score is calculated as a weighted harmonic mean of the Precision and Recall values, so the best score is 1.00 and the worst is 0.00. As can be seen in the image, the model takes values very close to 1 for each category, which means that it works as expected.

Support–indicates the number of occurrences of the class in the data set. We can visualize that "Glioma" appears 300 times, "No Tumor" 405 times, "Meningioma" 306 times, and Pituitary Tumor 300 times.

```
[21] accuracy = np.sum(pred==y_test_new)/len(pred)
      print("Acuratetea pentru setul de testare: {:.2f}%".format(accuracy*100))

Acuratetea pentru setul de testare: 98.86%
```

Figure 3.19 – Accuracy check for the test set

It is very important that after training the model, we check its accuracy on the test set, which in the present situation is new to the model because it has not used these images at all in the training process. It can often happen that the accuracy for a model is very high for the training set, which the network is used to, but very low for the test set, indicating that the model can still classify the images it is used to very well, is unable to correctly classify new images. For the model I created, the accuracy is very good at 98.86%, which means that out of 1311 unseen images, 98.86% are correctly classified.

```
train_pred = model.predict(X_train)
train_pred = np.argmax(train_pred, axis=1)
Y_train_new = np.argmax(Y_train, axis=1)
print("Acuratetea pentru setul de antrenare: {:.2f}%".format(np.sum(train_pred==Y_train_new)/len(train_pred)*100))

Acuratetea pentru setul de antrenare: 99.91%
```

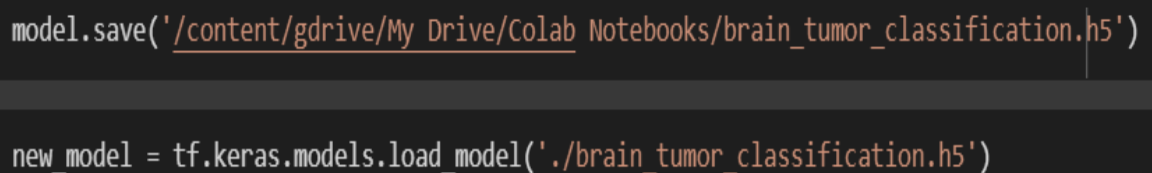
Figure 3.20– Accuracy check for the training set

Simply for informational purposes, I also wanted to check the accuracy for the training set. Obviously, the 99.91% accuracy is higher than for the test set because the network is already familiar with those images.

The fact that the accuracy is approximately the same value for the test set also indicates that the model does not overfit, so it has a high ability to classify images correctly.

3.4. Saving and loading the model

After a model is trained, it is important to save it so that its weights can be used later on other data sets without repeating the training, as this process can even take days for some networks. When publishing research models and techniques, most machine learning practitioners share: the code to build the model and the weights or parameters trained for the model (In this case, it's the `brain_tumor_classification.h5` file)



```
model.save('/content/gdrive/My Drive/Colab Notebooks/brain_tumor_classification.h5')  
  
new_model = tf.keras.models.load_model('./brain_tumor_classification.h5')
```

Figure 3.21 – Saving and loading the created model

In the figure above, the first line of code represents saving the created model, in the Colab Notebooks folder on Google Drive, and the second represents loading its weights so that it can be used. The imported TensorFlow library allows the save function to be used `save()` and loading function `tf.keras.models.load_model`.

3.5. Creating a button to upload an image from your car and predict it

To test the applicability of the created algorithm to specific images, chosen by the user, we created a button for uploading the images and a button that, once pressed, can provide a diagnosis about the images.

```

def img_pred(upload):
    for name, file_info in uploader.value.items():
        img = Image.open(io.BytesIO(file_info['content']))
        opencvImage = cv2.cvtColor(np.array(img), cv2.COLOR_RGB2BGR)
        img = cv2.resize(opencvImage, (224, 224))
        img = img.reshape(1, 224, 224, 3)
        p = new_model.predict(img)
        p = np.argmax(p, axis=1)[0]

        if p==0:
            p='GLIOMA'
        elif p==1:
            p='FARA TUMORA'
        elif p==2:
            print('Se preconizeaza ca este MENINGIOMA')
        else:
            p='TUMORA PITUITARA'

    if p!=2:
        print(f'Se preconizeaza ca este {p}')

```

Figure 3.22 – Creating a function that allows uploading pictures from the local machine

First, a function must be created to allow images to be loaded from the local machine, as shown in Figure 3.20. However, within this function, the images must be converted back to the numpy array so that the created model can work with them. Also, the images must be resized to the shape required by the model, namely 224 x 224. The variable *p* will help predictions on pictures, as it will take one of the values specific to each class.

```

uploader = widgets.FileUpload()
display(uploader)
button = widgets.Button(description='Predictie')
out = widgets.Output()
def on_button_clicked(_):
    with out:
        clear_output()
        try:
            img_pred(uploader)
        except:
            print('Imagine invalida/Nu s-a incarcat nicio imagine')
button.on_click(on_button_clicked)
widgets.VBox([button,out])

```

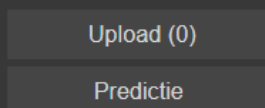


Figure 3.23 – Creating the upload button and the prediction button

The method that I considered the easiest in the present situation is to add some widgets available in the Python language by importing the package *ipwidgets*. These are graphical user interface (GUI) elements that help interact with the application you are building. The first two lines of code in Figure 3.21 are used to create the first button, the one for uploading pictures, called Upload, and the following code sequences deal with creating the Prediction button which, once pressed, after uploading, will provide a diagnosis for that picture .

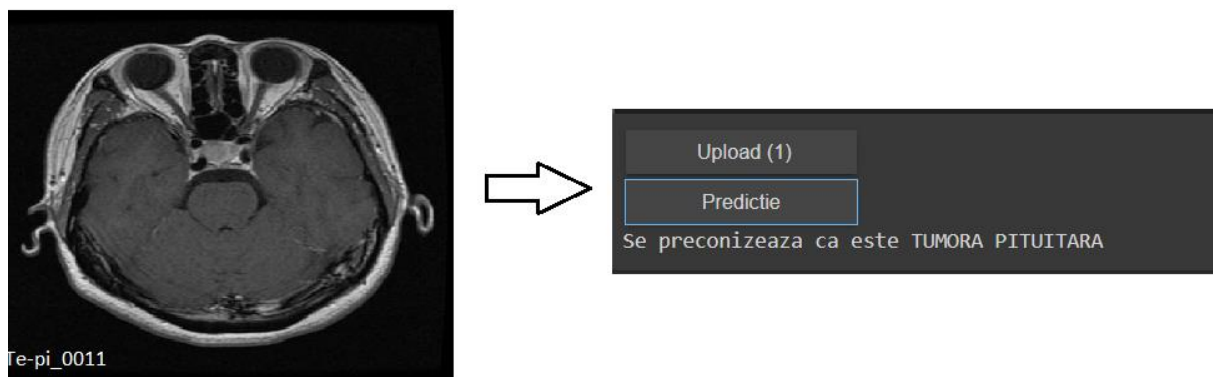


Figure 3.24 – Example of prediction for an image indicating a pituitary tumor

3.6. Problems encountered in creating the algorithm

3.6.1. Implementation issues

To implement the algorithm we used Google Colab created by Google Research which allows all users, whether experienced or not, to execute codes in the Python language, without the need to install other programs or packages on the local machine, because it works in browser and already comes with many Python packages installed. Since this is a cloud-based tool, the big advantage I've considered is that I don't need anything extra installed on my local machine, which is important for those who don't have hardware resources with enough memory.

One of the implementation issues in this case was the very long time to train the network, and as a solution I chose to use the GPU run instead of the CPU run.

Running on the GPU is faster than the CPU because the GPU contains more cores and this fact allows more processes to run in parallel. Initially, using the CPU, the model took 2 hours to complete only half of the set 50 epochs, but with the switch to the GPU, the model was fully trained in about an hour. The main impediment to using GPU in Google Colab is that at some point a usage limit is reached and you have to wait 12 hours before you can use it again. However, once the created model is trained, it can be saved and reused for predictions, and this aspect does not require the use of the GPU.

3.6.2. Creating networks that do not provide the desired results

In the case of convolutional neural network algorithms, there are many parameters that can take a lot of different values and that can help the model work better, or on the contrary, can cause it to work poorly. Unfortunately, although there are many resources available now, there is no well-established "recipe" that fits all types of images. In addition, it is known that in the case of medical pictures, especially MRI images, predictions are quite difficult. So, the only solution is to educate yourself and try various methods until you find the solution that gives the best results. The drawback is that this process is time- and resource-consuming, and is something to consider, especially since training networks can take so long.

Until reaching the final form of the presented algorithm, we tried other methods of implementing the convolutional neural network that would be able to classify the images correctly.

- Building a convolutional neural network with two convolutional layers

```
[41] model1 = Sequential()

# Convolutional layer 1
model1.add(Conv2D(32,(3,3), input_shape=(224, 224, 3), activation='relu'))
model1.add(BatchNormalization())
model1.add(MaxPooling2D(pool_size=(2,2)))

# Convolutional layer 2
model1.add(Conv2D(32,(3,3), activation='relu'))
model1.add(BatchNormalization())
model1.add(MaxPooling2D(pool_size=(2,2)))

model1.add(Flatten())

# Neural network

model1.add(Dense(units= 252, activation='relu'))
model1.add(Dropout(0.2))
model1.add(Dense(units=252, activation='relu'))
model1.add(Dropout(0.2))
model1.add(Dense(units=4, activation='softmax'))

optimizer = tf.keras.optimizers.Adam(learning_rate=0.001, decay=0.0001, clipvalue=0.5)
model1.compile(loss='categorical_crossentropy', optimizer='Adam', metrics=['accuracy'])
```

Figure 3.25– Construction of a convolutional neural network with two convolutional layers

In the figure above, we tried to build a classic convolutional neural network model. The first time, we chose to make the model sequential by function *Sequential()*, because this is the easiest way to build a model in Keras, as it allows building the model layer by layer through the function *add()*. Also, in the creation of convolutional layers, the *Conv2D* parameter is used, which sets the number of filters (32) with which the convolutional layers will learn and their size (3,3). It is used to detect features such as edges or curves in an image. Also appearing is the "ReLU" activation function, described in subsection 2.3.1, which is a classic activation function used in most convolutional neural network models.

Also, the pooling layer is created by the *MaxPooling2D* parameter that samples the images using the Max Pooling method, compared to the final model that uses Average Pooling. This method takes the maximum value from an input window, of size defined by *pool_size*. In the present case, this dimension takes the value (2, 2).

It is used *BatchNormalization()* to fine-tune the learning process and reduce the number of epochs the networks would need to learn.

After building the two convolutional layers, the network is created by building the fully connected layers through the Dense and Dropout functions, described in point 3.2.

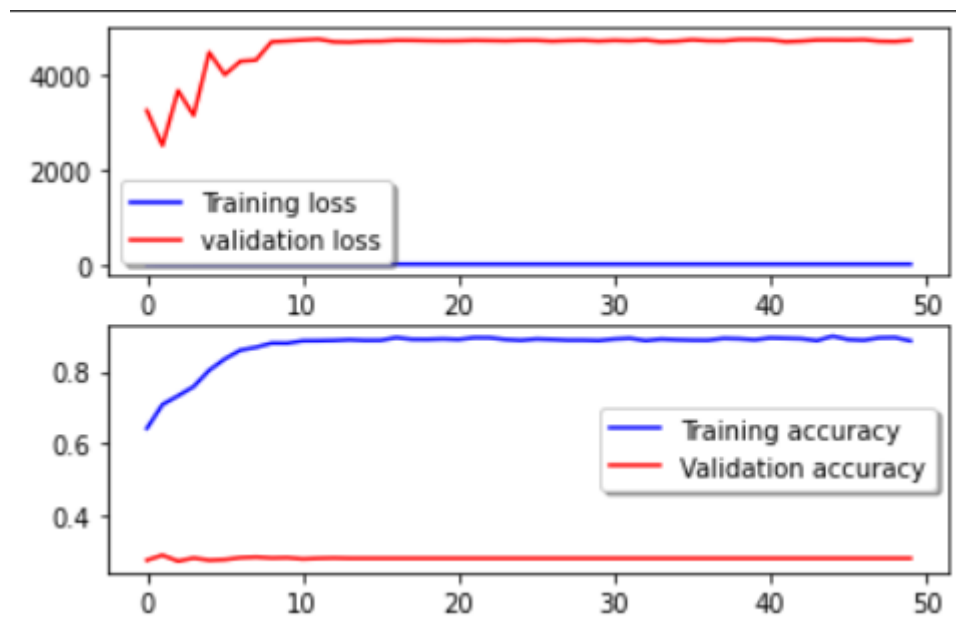


Figure 3.26 – Loss function and accuracy evolution for the convolutional two-layer model

As can be seen, the model does not seem stable and seems to produce overfitting, as the accuracy for the training set is quite high and the loss quite small and constant (blue), but for the validation set, things are completely the opposite. The accuracy is very low, showing no evolution, and the loss is very high and becomes constant from the 10th iteration (red).

```
[ ] accuracy = np.sum(pred==y_test_new)/len(pred)
    print("Acuratetea pentru setul de testare: {:.2f}%".format(accuracy*100))

Acuratetea pentru setul de testare: 33.18%

[ ] print(classification_report(y_test_new,pred))
```

	precision	recall	f1-score	support
0	0.00	0.00	0.00	300
1	0.32	0.96	0.48	405
2	0.57	0.15	0.24	306
3	0.00	0.00	0.00	300
accuracy			0.33	1311
macro avg	0.22	0.28	0.18	1311
weighted avg	0.23	0.33	0.20	1311

Figure 3.27 - Evaluation of the two-layer convolutional model

It can be observed that this kind of approach does not work very well for the data set used, since the accuracy for the test set is quite low: 33.18%. This means that out of the 1311 unseen images in the test set, only 33.18% of them can be classified correctly. Also, the values in the classification ratio do not indicate a very good performance either, most of them being much closer to 0 than to 1. As can be seen from the precision column, the number of correctly classified positive cases is quite small, the best being classified the images in the "Meningioma" category (0.57) probably due to the regular shape and well-defined edges for tumors of this kind.

- Building a convolutional neural network with six convolution layers

In the process of building a viable relationship, I tried increasing the number of convolutional layers and changing some parameters to see if I could get better accuracy on the test set. Basically, I kept the same build logic, but just added more layers.

```
model5 = Sequential()

# Convolutional layer 1
model5.add(Conv2D(64,(7,7), input_shape=(224, 224, 3), padding='same', activation='relu'))
model5.add(BatchNormalization())
model5.add(MaxPooling2D(pool_size=(2,2)))

#Convolutional layer 2
model5.add(Conv2D(128,(7,7), padding='same', activation='relu'))
model5.add(BatchNormalization())
model5.add(MaxPooling2D(pool_size=(2,2)))

# Convolutional layer 3
model5.add(Conv2D(128,(7,7), padding='same', activation='relu'))
model5.add(BatchNormalization())
model5.add(MaxPooling2D(pool_size=(2,2)))

# Convolutional layer 4
model5.add(Conv2D(256,(7,7), padding='same', activation='relu'))
model5.add(BatchNormalization())
model5.add(MaxPooling2D(pool_size=(2,2)))

# Convolutional layer 5
model5.add(Conv2D(256,(7,7), padding='same', activation='relu'))
model5.add(BatchNormalization())
model5.add(MaxPooling2D(pool_size=(2,2)))

# Convolutional layer 6
model5.add(Conv2D(512,(7,7), padding='same', activation='relu'))
model5.add(BatchNormalization())
model5.add(MaxPooling2D(pool_size=(2,2)))

model5.add(Flatten())

# Full connect layers

model5.add(Dense(units= 1024, activation='relu'))
model5.add(Dropout(0.25))
model5.add(Dense(units=512, activation='relu'))
model5.add(Dropout(0.25))
model5.add(Dense(units=4, activation='softmax'))

model5.compile(optimizer=SGD(learning_rate=0.001), loss='categorical_crossentropy',
               metrics= ['categorical_accuracy'])
```


Figure 3.28 - Building a new model with six convolutional layers

As can be noted from Figure 35, six layers are created of which the first layer has 64 filters of size (7, 7), the second and third have 128 filters of the same size, the fourth and fifth layer have 256 of filters, and the sixth 512, also of the same size.

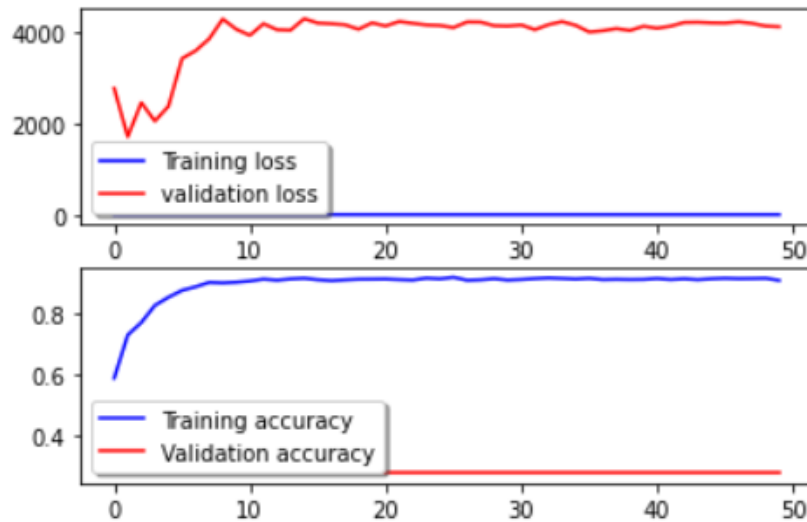


Figure 3.29 – Evolution of the loss function (top) and accuracy (bottom) for the six-layer convolutional model

It seems that for this model too, the same problem occurs. The model does not seem consistent, being for him also a great risk of overadapting. The accuracy on the training set (blue) is very high, and for the validation set it is very low (red), and conversely the loss function, which does not indicate a correct operation of the algorithm at all.

```

▶ pred = model5.predict(X_test)
  pred = np.argmax(pred,axis=1)
  y_test_new = np.argmax(Y_test,axis=1)

[ ] accuracy = np.sum(pred==y_test_new)/len(pred)
  print("Accuracy on testing dataset: {:.2f}%".format(accuracy*100))

Accuracy on testing dataset: 30.89%

[ ] print(classification_report(y_test_new,pred))

```

	precision	recall	f1-score	support
0	0.00	0.00	0.00	300
1	0.31	1.00	0.47	405
2	0.00	0.00	0.00	306
3	0.00	0.00	0.00	300
accuracy			0.31	1311
macro avg	0.08	0.25	0.12	1311
weighted avg	0.10	0.31	0.15	1311

Figure 3.30 – Evaluation of the six-layer convolutional model

The accuracy tested on the test set is even lower for this model, having the value of 30.89%, despite the fact that it has more layers, which in some situations can help a lot to achieve good accuracy. And the classification ratio shows even lower values, most of which are 0. The only class that this algorithm can identify very little is the class specific to non-tumor images.

So, comparing the three models described in this paper, it is clearly observed that the algorithm using the weights of the EfficientNetB0 model is much better and works correctly.

As I mentioned, there is no general solution for such algorithms, so what works for some images will not work for others. Also, there are many ways in which you can achieve favorable results, but often time and resources are limited, because only by testing them can you check whether the model really works or not.

Conclusion

The application created with the help of convolutional neural networks has the ability to identify if a person has a brain tumor and classify the type that person has (glioma, meningioma, or pituitary tumor), based on magnetic resonance images.

To create the application, the best method was to use convolutional neural networks, since they are most often used for image classification. As you can see, multiple attempts are needed to create the "perfect" algorithm, that's why there are numerous studies done by specialized people who approach the solution of certain medical problems by means of artificial intelligence.

Convolutional neural networks have many advantages, and one of the most important reasons to choose their use, is the fact that they are very effective in reducing the number of parameters, without losing the quality of the models. These have even reached or exceeded in certain cases, human capacity, an aspect that cannot be overlooked in the reality of 2022, when technology is expected to merge with human power to achieve the expected goals.

The performance of RNN is due to the constant innovations that have been brought to it type of technology, for example the use of ReLu activation functions. Also, the evolution of storage and processing capabilities and power, such as the development of GPU processors, played an equally important role in their increasingly frequent choice of use [78]. Also, the possibility of reusing and adapting the models already trained by experts in the field to your own data set is a real advantage. Creating and training networks can be a time- and resource-consuming process, hence the existence of such well-known methods as EfficientNetB0, VGG16, etc. it allows the creation of valid and perfectly usable applications, even if the developer does not have such a large set of data. Therefore, the concept of Transfer Learning introduced through this technology brings real added value.

I am of the opinion that this kind of applications are topical and very necessary for the medical field. As seen in subchapter 1.2, brain tumors are increasingly common in people everywhere for various reasons, and it is important that they are identified early. Although doctors have conventional methods of identifying brain tumors, the work they have to do is currently manual.

I believe that the introduction of an application that automatically identifies whether the patient has a tumor or not would be of great help to both doctors and patients, as it would save an enormous amount of time and allow doctors to focus more , perhaps, on the patient's treatment plan. Obviously, there can be errors on both sides, both in the case of applications and in the case of human diagnostics, but I think that combining the two resources would have a minimal chance of failure.

Also, the created application can obviously be improved. The model proposed by me is only a prototype. One of the improvements would be to train on a much larger data set to get the best results. If the data set were much larger, I believe that this aspect would also allow the use of other types of higher performing architectures with more layers to improve the results.

Also, the created model could in the future be introduced into an application with a much friendlier interface. An application can be created on the desktop or on the mobile that can be handy for both doctors and patients. It should be noted, however, that if it were to be used by a patient, he or she should still confirm the result with a medical professional.

Last but not least, the range of diagnostics that the application could offer can be increased. For example, it could be enhanced to identify the location of brain damage from various accidents, or perhaps even capture the presence of blood clots in the brain. Another added value that can be brought within the application, although at the moment it is still very difficult, would be to identify the stage of tumor evolution, but I believe that for this, it would be necessary to perfectly combine the expertise of doctors with expert engineers in the development Deep Learning algorithms.