Name: Umit Etlik
Mail : umit.etlik@student.uclouvain.be
Github : https://github.com/umit-commits

Tumor Detection using Deep learning model

Contents

Introduction

As we delve into the complexity of tumor detection, it becomes evident that the existing methods often present challenges. Traditional scans may overlook small tumors, and the time lag in obtaining results can be critical, especially in urgent medical situations. This discrepancy underscores the need for a more efficient and timely solution. Enter deep learning, a cutting-edge approach in artificial intelligence. By leveraging the power of deep learning, we aim not only to improve the accuracy of tumor detection but also to expedite the process, potentially saving lives through swift and reliable identification of brain tumors. This technology's ability to analyze vast datasets and recognize intricate patterns offers a promising avenue for revolutionizing healthcare, ensuring that interventions are quick and outcomes are optimized.

Data Collection

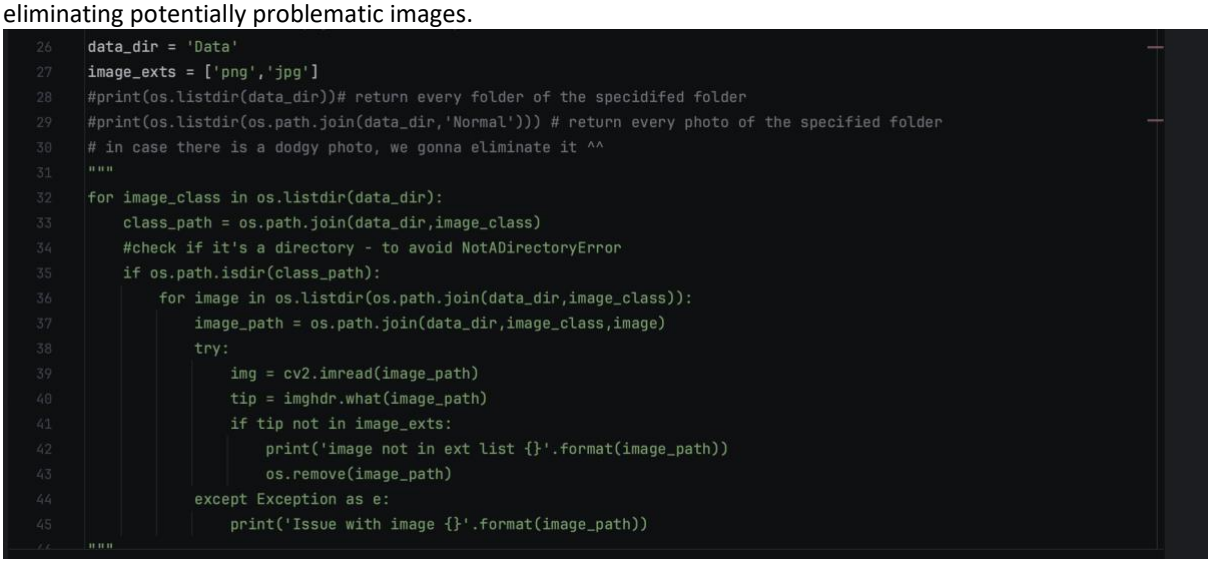
I have collected the thousands of MRI images from Kaggle. For each tumor type data (Meningioma, Glioma, Pituitary tumor) and normal brain MRI images I have been able to store them in different folders. It is important to note that data should be shuffled in order to have our model work properly.

Model Architecture

Data processing

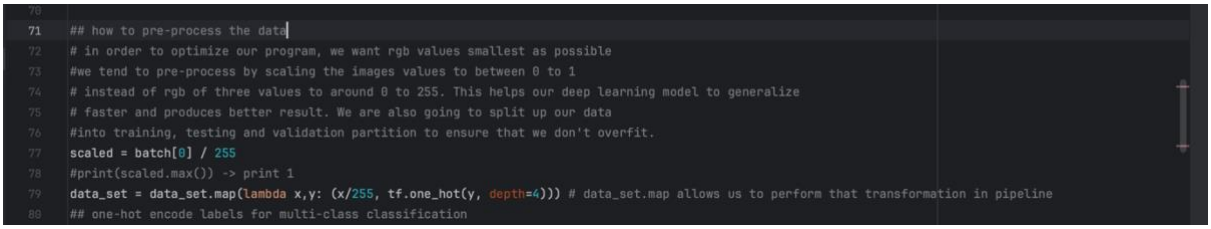
- Data-loading : **tf.keras.utils.image_dataset_from_directory** for efficient loading, employed a numpy iterator for batch-wise access to data.

- Checked and removed images with non-standard extensions. This step ensures data integrity by eliminating potentially problematic images.



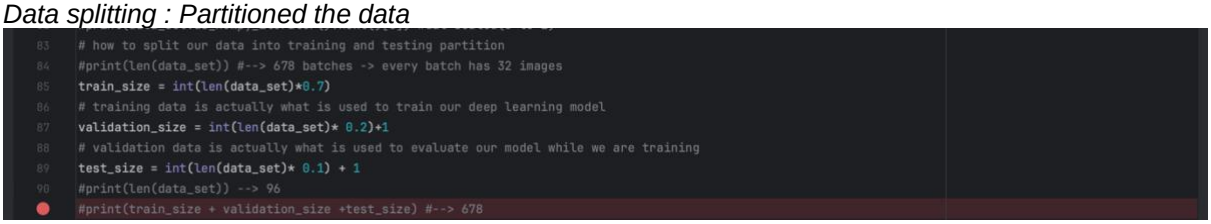
```
data_dir = 'Data'
image_exts = ['png','jpg']
#print(os.listdir(data_dir))# return every folder of the specidifed folder
#print(os.listdir(os.path.join(data_dir,'Normal'))) # return every photo of the specified folder
# in case there is a dodgy photo, we gonna eliminate it ^^
"""
for image_class in os.listdir(data_dir):
    class_path = os.path.join(data_dir,image_class)
    #check if it's a directory - to avoid NotADirectoryError
    if os.path.isdir(class_path):
        for image in os.listdir(os.path.join(data_dir,image_class)):
            image_path = os.path.join(data_dir,image_class,image)
            try:
                img = cv2.imread(image_path)
                tip = imghdr.what(image_path)
                if tip not in image_exts:
                    print('image not in ext list {}'.format(image_path))
                    os.remove(image_path)
            except Exception as e:
                print('Issue with image {}'.format(image_path))
"""
```

- **Data Scaling and Formatting:** Scaled pixel values to a range between 0 and 1 for better convergence during training. Applied one-hot encoding to categorical labels for multi-class classification.



```
## how to pre-process the data
# in order to optimize our program, we want rgb values smallest as possible
#we tend to pre-process by scaling the images values to between 0 to 1
# instead of rgb of three values to around 0 to 255. This helps our deep learning model to generalize
# faster and produces better result. We are also going to split up our data
#into training, testing and validation partition to ensure that we don't overfit.
scaled = batch[0] / 255
#print(scaled.max()) -> print 1
data_set = data_set.map(lambda x,y: (x/255, tf.one_hot(y, depth=4))) # data_set.map allows us to perform that transformation in pipeline
## one-hot encode labels for multi-class classification
```

- *Data splitting : Partitioned the data*



```
# how to split our data into training and testing partition
#print(len(data_set)) #--> 678 batches -> every batch has 32 images
train_size = int(len(data_set)*0.7)
# training data is actually what is used to train our deep learning model
validation_size = int(len(data_set)* 0.2)+1
# validation data is actually what is used to evaluate our model while we are training
test_size = int(len(data_set)* 0.1) + 1
#print(len(data_set)) --> 96
#print(train_size + validation_size +test_size) #--> 678
```

into training, validation and testing sets.

Model Building

- *Model architecture :* Defined a sequential model using Keras, added convolutional layers, batch normalization, max-pooling, dropout, and dense layers, utilized softmax activation for multi-class classification.

```
model = Sequential()

# Add layers to the model
model.add(Conv2D( filters: 16,  kernel_size: (3, 3),  strides: 1, activation='relu', input_shape=(256, 256, 3)))
model.add(BatchNormalization()) #Batch Normalization helps stabilize and accelerate the training process by normalizing the input to each layer.
model.add(MaxPooling2D()) # Max pooling is typically used to reduce the spatial resolution while retaining the most salient features.
model.add(Dropout(0.25)) # reduce overfitting

model.add(Conv2D( filters: 32,  kernel_size: (3, 3),  strides: 1, activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D())
model.add(Dropout(0.25))

model.add(Conv2D( filters: 16,  kernel_size: (3, 3), strides=1, activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D())
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense( units: 256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense( units: 4, activation='softmax'))
#model.add(Dense(3, activation='softmax'))
# softmax is used for multi-class classification, and it provides probabilities for each class
#optimasirs
model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])
lr_callback = LearningRateScheduler(lr_scheduler)

model.summary()
```

- *Learning Rate Scheduler: Implemented a learning rate scheduler to dynamically adjust the learning rate during training. It contributes to better training stability.*

```
1 usage
def lr_scheduler(epoch, lr):
    if epoch % 10 == 0 and epoch > 0:
        lr = lr * 0.9
    return lr
```

Model Training

- I used the **fit** method to train the model, specifying the number of epochs and incorporating validation data. I then integrated callbacks for TensorBoard logging and model checkpointing in order to save model_weights so I don't have to launch training every time i analyse my model.

```
# TRAIN
#Tensorboard will store the logs and information about the training process in 'logs'
logdir = 'log1'
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=logdir)
checkpoint_callback = ModelCheckpoint( filepath: 'model_weights.h5', save_best_only=True)
history = model.fit(train,epochs=20, validation_data=validation,callbacks=[tensorboard_callback,checkpoint_callback])
model.load_weights('model_weights.h5')
#loss vizualisation
```

- *Visualization of Training Metrics:* I plotted training and validation loss, as well as accuracy, over epochs for visual analysis and performance assessment.

```
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 4))

axes[0].plot(history.history['loss'], color='teal', label='train_loss')
axes[0].plot(history.history['val_loss'], color='orange', label='val_loss')
axes[0].set_title('Loss', fontsize=20)
axes[0].legend(loc="upper left")

# visualize Accuracy
axes[1].plot(history.history['accuracy'], color='teal', label='train_accuracy')
axes[1].plot(history.history['val_accuracy'], color='orange', label='val_accuracy')
axes[1].set_title('Accuracy', fontsize=20)
axes[1].legend(loc="upper left")

plt.show()

#########
```

Model Evaluation

- *Loading pre-trained model : Loading the trained model from saved weights for evaluation on unseen data(data that I have reserved for testing → test_size)*

```
loaded_model = load_model('model_weights.h5')
```

- *Performance : I have used performance metrics such as precision, recall and accuracy metrics to evaluate my model's performance.*

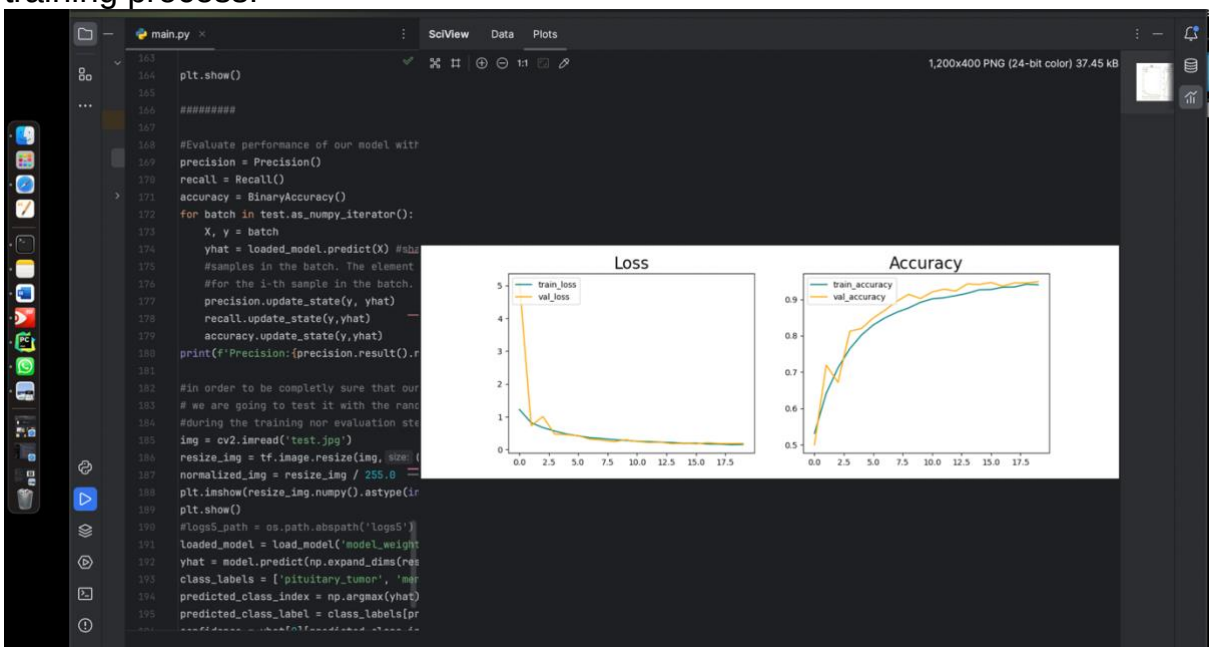
```
#Evaluate performance of our model with partition data_set that he hasn't seen yet
precision = Precision()
recall = Recall()
accuracy = BinaryAccuracy()
for batch in test.as_numpy_iterator():
    X, y = batch
    yhat = loaded_model.predict(X) #shape of our yhat will be "(batch_size, num_classes), where batch_size is the number of
    #samples in the batch. The element of the yhat[i,j] will represent the predicted probability of the j-th class
    #for the i-th sample in the batch.
    true_labels.extend(np.argmax(y, axis=1))
    predicted_labels.extend(np.argmax(yhat, axis=1))

    precision.update_state(y, yhat)
    recall.update_state(y,yhat)
    accuracy.update_state(y,yhat)
print(f'Precision:{precision.result().numpy()}, Recall:{recall.result().numpy()}, Accuracy:{accuracy.result().numpy()}')
```

- **Confusion Matrix:* Constructed and visualized a confusion matrix to gain insights into the model's classification performance across 4 different classes.

```
class_labels = ["glioma","menin","normal","pitu"]
num_classes = len(class_labels)
conf_matrix = np.zeros( shape: (num_classes, num_classes), dtype=int)

for true_label, predicted_label in zip(true_labels, predicted_labels):
    conf_matrix[true_label, predicted_label] += 1

plt.imshow(conf_matrix, interpolation='nearest', cmap=plt.cm.Blues)

plt.colorbar()

tick_marks = np.arange(num_classes)
plt.xticks(tick_marks, class_labels, rotation=45)
plt.yticks(tick_marks, class_labels)

for i in range(num_classes):
    for j in range(num_classes):
        plt.text(j, i, str(conf_matrix[i, j]), ha="center", va="center", color="w")

plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.title('Confusion Matrix')

plt.show()
```

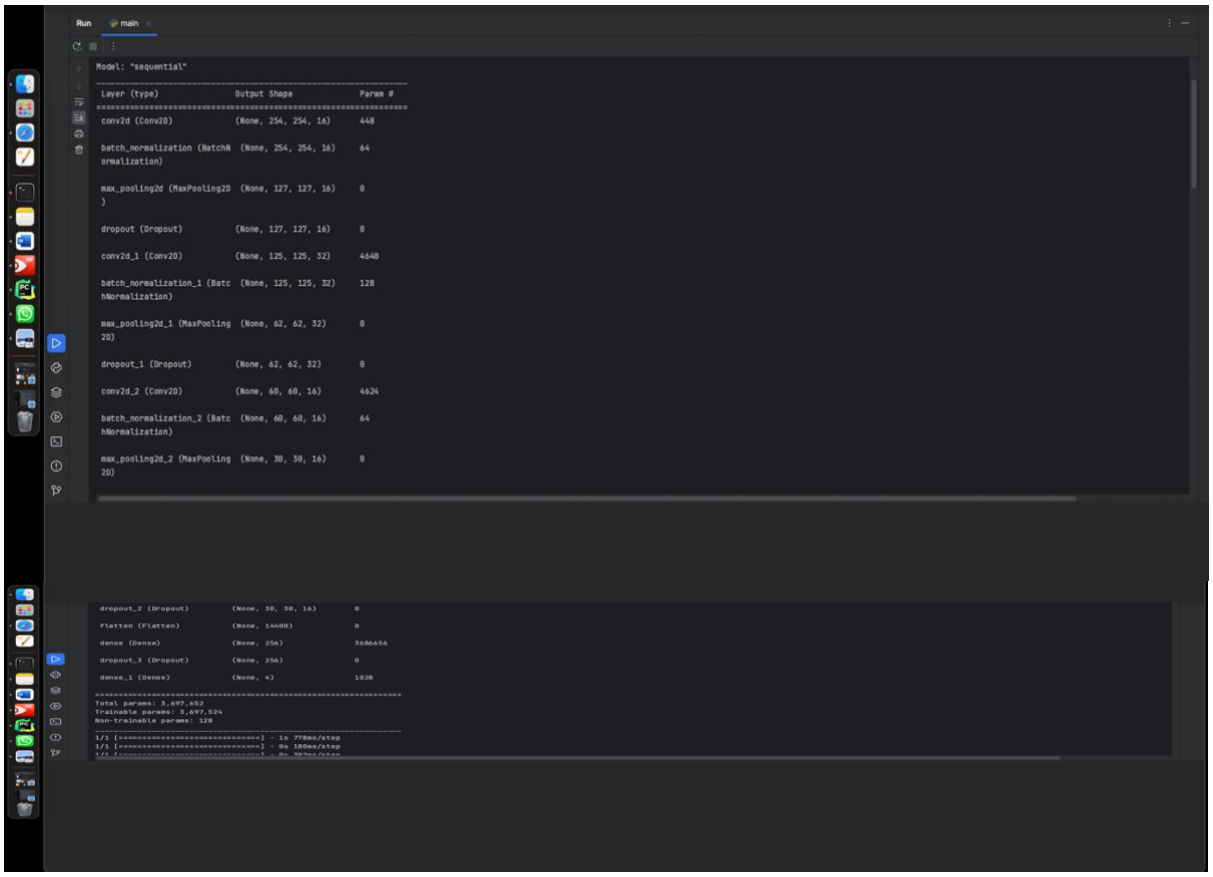
Result

Visual analysis of evaluation of my model's accuracy and loss during the training process.

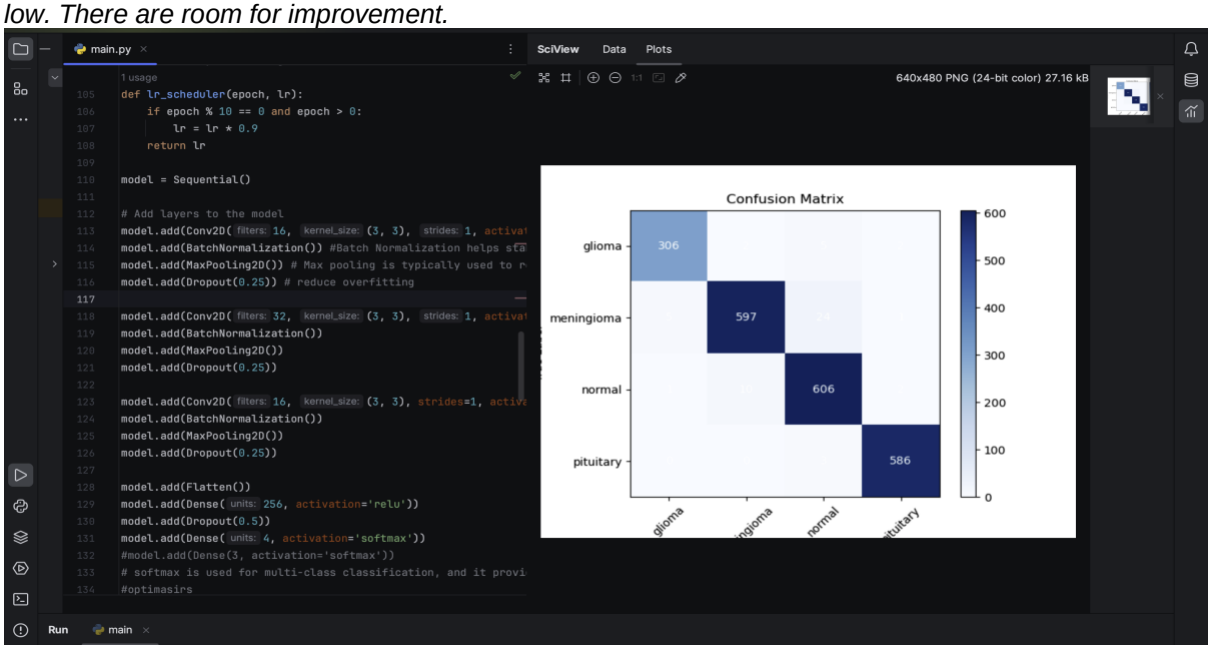


*Output shape (*batch_size, height, width, channels). It specifies the shape of the output tensor after each layer. Reduction in shape size is a consequence of using convolutional and pooling layers, which are designed to progressively reduce spatial dimensions while capturing and emphasizing important features in the data.

3,697,524 trainable parameters and 128 non-trainable parameters which is a consequence of batch normalization layers.



Confusion Matrix, as it appears, the intersection of true label and predicted label for glioma tumor is low. There are room for improvement.



Precision : 0.98
Recall : 0.97
Accuracy : 0.99

Challenges

- *With a given input test in which MRI image contains tumor, the predicted values for 3 classes* (Meningioma, Glioma, Pituitary tumor) are often close. I tried to adjust the model architecture and considered modifying class weights.

- *Some issues in the code, such as importing libraries; I was having hard time while importing keras on pycharm. I fixed the problem by searching the solution in forums. I couldn't either import Seaborn library to create confusion matrix where it would take only few lines.*

- *Incorrect Prediction: While my module goes through the test partition quite promising, with given the input image, incorrect prediction happens. It might be due to the fact that recognition of tumor's type is complex and it needs more dataset.*