CS 122

Dr. Wendy Lee

Team Eagle - Friend Dickinson, Matthew Fu, Bella Wei

Project Report

Detecting and Categorizing Brain Tumors With Python Using a CNN

I. Introduction

For our project, we will use the Kaggle Brain Tumor MRI dataset to classify brain tumors. The reason that we have chosen this dataset is that Friend's godfather passed away from a brain tumor. We believe that being able to quickly and accurately detect the occurrence of a brain tumor would allow doctors to treat them sooner. Our goal in this project is to be able to develop a convolutional neural network (CNN) that can accurately detect the type of tumor given an image of a human brain MRI. We want to be able to detect the occurrence of a brain tumor by designing and training a neural network that can classify brain tumors with as much accuracy as possible without overfitting to the particular dataset we've chosen.

Before we get into the details of our project, first, some medical context. Tumors are abnormal masses of tissue that form when cells divide more times than they should. Not all tumors are cancerous. Some are benign, meaning they may grow large, but they won't spread into nearby tissues, while others are malignant, meaning they will. The former type of tumor is known as cancer, which is the second leading cause of death in the United States. Cancer needs to be detected and classified as early into its development as possible in order to give doctors the best chance of treating it, giving us quite the motivation to be able to classify images of tumors.

There are many ways doctors check for tumors in the human body, from touch-based methods when checking for breast or testicular cancer, or X-ray machines, which can help doctors look for tumors in the kidneys, stomach, lungs, or bones. For tumors in the brain, however, MRI (magnetic resonance imaging) or CT (computed tomography) scans are typically used. Doctors look at such images and use their knowledge of what areas should and shouldn't be bright depending on the material of that area of the skull, forming an educated guess as to whether what they're looking at is a tumor at all, and if it is a tumor, what type it is. This is an arduous task that requires lots of experience in the field to accomplish, making it a perfect target for automation using machine learning.

II. Methods and Materials

While image processing and neural networks can be implemented in many different languages, python is typically the go-to for projects of this nature and as this is for our Python course, that works perfectly. Python has many packages designed to help with machine learning and image processing. The packages that we decided to use for this project are tensorflow, matplotlib, numpy, pydot, and keras. From the keras machine learning library, we used the Conv2D, BatchNormalization, Activation, MaxPooling2D, GlobalAveragePooling2D, Dropout, and Dense layering functions to help generate our neural network model. We used matplotlib to plot our images.

We chose the dataset Brain Tumor MRI Dataset from Kaggle(See figure 1). The Kaggle link is https://www.kaggle.com/masoudnickparvar/brain-tumor-mri-dataset. The usability of this dataset is 7.5 which means the dataset is easy to understand with the machine-readable file format. The dataset contains 7022 images of human brain MRI images which are classified into four classes: glioma, meningioma, no tumor, and pituitary. The dataset also comes with 78% for model training and the rest for model testing.

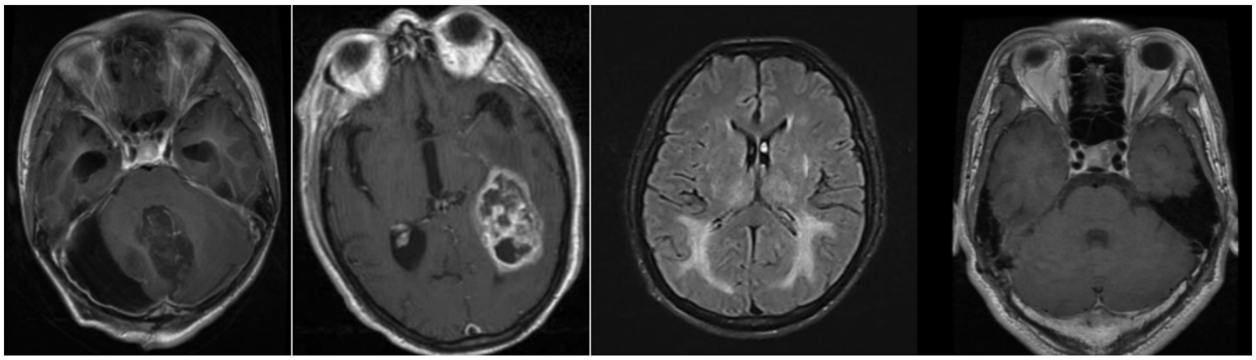

Figure 1: Sample Data (left to right) Glioma, Meningioma, No tumor, Pituitary

The model that we decided to use for this project is a 2D convolutional neural network or CNN for short. We thought this would be the best model for this project because of how effective CNN's are at image classification, the most notable example of its effectiveness at image classification being the hand-writing dataset.

We first set our image size to 64 by 64 and batch size to 32. We loaded our data from two folders, one for training and one for testing, and placed them into variables using the image_dataset_from_directory method from keras.preprocessing. We split the Training data into two parts, one for training our model and the other for validating it. We used a random seed to randomize which training images go into which part. We use all the images from the Testing folder and put them into one section. We grayscale all the images to set the channel depth to 1.

We created a list of the four class names for plotting purposes. We plotted a few images of each class so that we can get a rough idea of what each tumor looked like.

To prevent overfitting, we first augment our data. We create a Sequential object that contains the RandomFlip method and RandomRotation method from Keras. These methods will randomly flip and rotate the images respectively. Once the images have been augmented, we then pass them into the entry block.

For our entry block, we first rescale the input from the [0, 255] range to the [0,1] range. Afterwards, we pass our input first into a Conv2D layer. We normalize the output of our Conv2D layer with the BatchNormalization method, which has the effect of reducing the number of epochs required to train our model. Finally, we call the ReLU activation function using the Activation method so that our model can learn from the output of the BatchNormalization layer. We repeat this process again, except that we change the layer's dimension from 32 by 32 to 64 by 64. Prior to putting our model into layers of larger dimensions, we save a layer as the residual, which we will add at the end of every iteration of the larger layers.

For layers of size 128, 256, 512, and 728, instead of using Conv2D, we used SeparableConv2D. This replacement has the effect of making our model faster and perform more accurately than if we used Conv2D. Before calling SeparableConv2D, we call Activation so the layer can learn . Similar to our entry block, after calling SeparableConv2D, we call BatchNormalization to normalize the layer. After repeating the processing for a second time, keeping the size the same, we call the MaxPooling2D method. This method takes the largest value of each rectangular block of the previous layer. We call this method to make the model more translation invariant, so that the model can identify the tumor or non-tumor correctly. Finally, at the end of the iteration, we add our residual layer, and set the current layer that we have as our next residual.

We call SeparableConv2D with size of 1024, followed by BatchNormalization and Activation after the final iteration. Then we call the GlobalAveragePooling2D, which reduces the spatial dimension of the layer to 1. Because we have four classes, we set the activation function to softmax. We add a Dropout layer to prevent any kind of overfitting before adding a Dense layer, which connects all the layers together in order to be able to classify images.

We train the model over 50 epochs, which only took about 5 minutes in total (running locally). We call the compile method, setting the optimizer to the Adam algorithm, the loss function to categorical crossentropy, and our metrics to accuracy. We finally train our model by calling the fit method.

III. Results

We tested our model using unseen images from the Testing folder, which importantly were NOT used to train the model and were kept untouched until now. Our model predicted the pictures with 93% accuracy. To test exactly how accurate our model was at predicting brain tumors, we generated a confusion matrix using our model and tensorflow's built in confusion matrix function. This matrix shows predicted labels (x-axis) vs. true labels (y-axis) for every image in the testing set, with % of the total of that label mapped to the darkness of the blue color for each respective cell in the matrix (Figure 2).

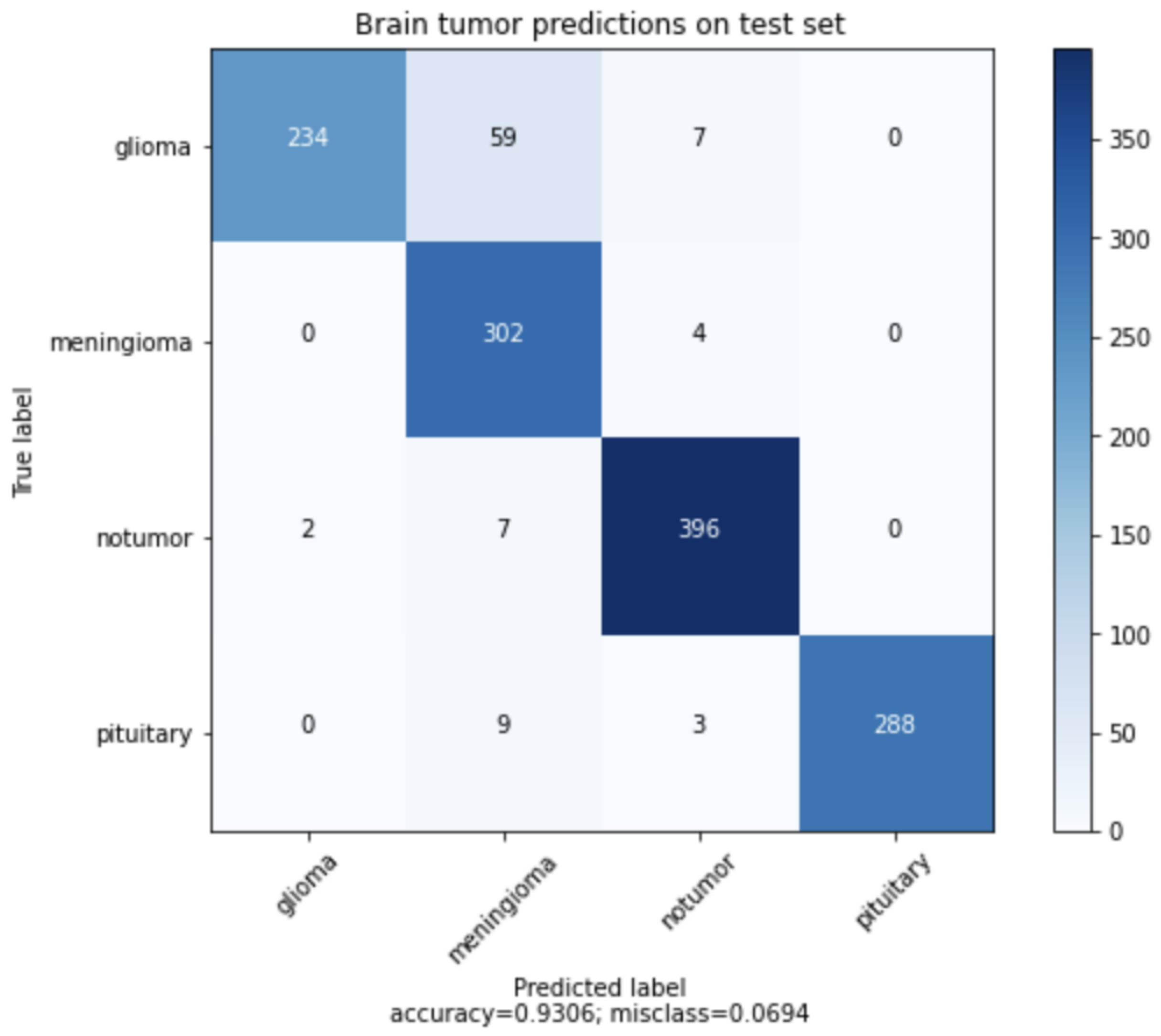

Figure 2: Confusion matrix showing 93% model accuracy on test set

We found that the overall accuracy was 93%, and there were only 7% misclassified images. The most amount of misclassifications came from classifying meningioma. The model got confused with glioma the most when attempting to classify an image as meningioma. This most likely resulted from the high similarity between meningioma and glioma images.

We finally displayed a random few of the images that the model failed to accurately predict as examples (Figure 3).

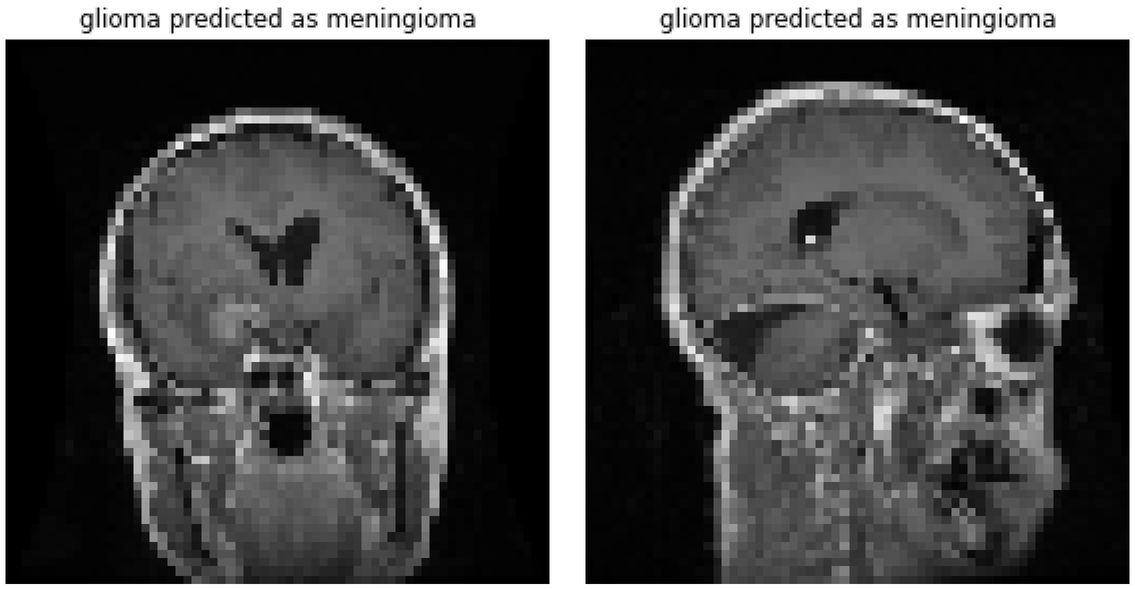
glioma predicted as meningioma
glioma predicted as meningioma

Figure 3: Some misclassified tumor images

IV. Discussion

Some challenges that we ran into during the project were related to neural network design, as a lot of it seemed very abstract and like something you just need lots of experience with for it to make total sense - but we ended up with a decent model in the end. Another challenge was getting the GPU support for Tensorflow set up. Thankfully, these challenges didn't amount to much delay in the end.

Our model didn't take very long to train, and still ended up with a fairly high predictive power - at least better than someone with little training in the art of identifying tumors could match. We achieved our original goal of creating a neural network based program that could identify tumors, and learned a lot about python, image processing, and machine learning along the way. If this was a more serious research project, the next steps would be clear: first, we would want to get a longer training session with these exact settings, maybe hours or days long. Then, we could try tweaking some parts of our model's setup and attempt to come up with as close to 100% accuracy as possible, although "close" is as far as any machine learning program will ever be able to get. Another important step would be trying it on newly generated pictures of tumors using similar imaging methods to see if our model retains its predictive power as well as it did on this dataset.

V. How to Run

Tested on Windows 10 with an Nvidia GPU:

- Download the term project notebook
- Download the kaggle brain tumor dataset:
  https://www.kaggle.com/masoudnickparvar/brain-tumor-mri-dataset
  - Put the folder in the same folder as project notebook
- Download python 3.7
- Download pip (python package manager)
  - In terminal, do "pip install [package name]" for every top level package in the imports cell of the project notebook
    - tensorflow, os, matplotlib, etc

    - (there might be some way to make this faster with a file with all the imports needed - but in this case there's not too many)
    - If you want increased speed by making tensorflow use your GPU, you need to follow these steps: https://www.tensorflow.org/install/gpu
- Download jupyter-lab (or some other way to run python notebooks)
- To start jupyter lab, go in the project folder in a terminal and run "jupyter-lab"
    - To run by training the CNN, leave the training cell (look for the cell with "model.fit" in it) uncommented and the "start from a checkpoint" cell commented and click Run->Run All Cells.
    - To start from a checkpoint, make sure you have the checkpoint file in a model_checkpoints directory and comment out the training cell, uncomment the checkpoint cell. (I set it to look for the checkpoint with the same number as the set number of epochs - by default 50. You can customize the filename if that isn't wanted)

Google Colab:

- Should be similar to above. Don't have to download python/pip or install packages, but may be slower. Epochs took about 5 seconds locally with GPU support enabled for comparison - if it turns out it's not TOO much slower Colab would definitely be the way to go.