

Brain Tumor MRI Binary Classification Study

**By**

**Hussein Kasim**

**Supervisor**

**Samer Nofal**

**Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Bachelor of Science in Computer Science**

**German Jordanian University**

**School of Electrical Engineering and Information Technology**

**Department of Computer Science**

**January 2023**

Authorization Form

I, Hussein Kasim, authorize the German Jordanian University to supply copies of my to libraries, establishments or individuals on request, according to the Regulations of the German Jordanian University

**Signature:** 

**Date: 02/01/2023**

Committee Decision

This Thesis **“Brain Tumor MRI Binary Classification Study”** was Successfully Defended and Approved on Click or tap to enter a date.

|  |  |
| --- | --- |
| **Committee Members** | **Signature** |
| **Supervisor** |  |
| **Examiner 1** |  |
| **Examiner 2** |  |
|  | ……………………… |

Dedication

I dedicate this work to my parents, professors, friends, and everyone else who supported me during my studies. Thank you for everything.

**Hussein Kasim**

Acknowledgments

In the name of Allah, I would like to praise Allah the Almighty, the most Gracious, the most Merciful, and the most Beneficent for His blessing given to me during my studies and in writing and completing this thesis. May Allah’s blessing go to His final Prophet Muhammad (Peace Be Upon Him), his family and companions.

I would also like to express my deepest gratitude to my respected supervisor, Dr. Samer Nofal, who made this work possible. His invaluable advice, support, and patience were key for me throughout my studies.

I am extremely grateful for the unwavering support of my parents and family members for allowing me the privilege of carrying out my studies at this esteemed institution, as well as their consistent encouragements during my studies.

I am thankful of the support of my friends and colleagues, who provided constant support over the course of my study period.

Finally, I would like to thank the examiners and committee members for their time and work.

**Hussein Kasim**

List of Tables

[**Table 5.1:** Experiments 1-12 Results 25](#_Toc124100167)

[**Table 5.2:** Experiments 13-24 Results 29](#_Toc124100168)

List of Figures

[**Figure 2.1:** An Example Image 2](#_Toc124100003)

[**Figure 3.1:** The ResNet Architecture Used 4](#_Toc124100004)

[**Figure 4.1:** Imported Modules 5](#_Toc124100005)

[**Figure 4.2:** Importing the Dataset 6](#_Toc124100006)

[**Figure 4.3:** Checking the Directories 6](#_Toc124100007)

[**Figure 4.4:** Loading the Data 7](#_Toc124100008)

[**Figure 4.5:** Calculating the Mean and Standard Deviation 8](#_Toc124100009)

[**Figure 4.6:** Experiments 1-12 Mean and Standard Deviation Values 9](#_Toc124100010)

[**Figure 4.7:** Experiments 13-24 Mean and Standard Deviation Values 9](#_Toc124100011)

[**Figure 4.8:** Data Augmentation and Normalization 10](#_Toc124100012)

[**Figure 4.9:** Data Loaded Into the CPU 10](#_Toc124100013)

[**Figure 4.10:** Device Dataloader 11](#_Toc124100014)

[**Figure 4.11:** Dataloaders Moved to the GPU 11](#_Toc124100015)

[**Figure 4.12:** Predictions Function 12](#_Toc124100016)

[**Figure 4.13:** Training Step Function 13](#_Toc124100017)

[**Figure 4.14:** Validation Step Function 13](#_Toc124100018)

[**Figure 4.15:** Validation Epoch End Step Function 14](#_Toc124100019)

[**Figure 4.16:** Epoch End Function 14](#_Toc124100020)

[**Figure 4.17:** Convolutional Block Function 15](#_Toc124100021)

[**Figure 4.18:** ResNet Class 16](#_Toc124100022)

[**Figure 4.19:** Model Moved to the GPU 16](#_Toc124100023)

[**Figure 4.20:** Helper Functions and Variables Setup 18](#_Toc124100024)

[**Figure 4.21:** Training and Validation Loop 19](#_Toc124100025)

[**Figure 4.22:** Initial Prediction 19](#_Toc124100026)

[**Figure 4.23:** Hyperparameter Values 20](#_Toc124100027)

[**Figure 4.24:** Fit Function 20](#_Toc124100028)

[**Figure 4.25:** Accuracy Plotting 21](#_Toc124100029)

[**Figure 4.26:** Error Rate Plotting 21](#_Toc124100030)

[**Figure 4.27:** Precision Plotting 21](#_Toc124100031)

[**Figure 4.28:** Recall Plotting 21](#_Toc124100032)

[**Figure 4.29:** Losses Plotting 22](#_Toc124100033)

[**Figure 4.30:** Singular Image Testing on the First Image 22](#_Toc124100034)

[**Figure 4.31:** Tumor Detected Example 23](#_Toc124100035)

[**Figure 4.32:** No Tumor Detected Example 23](#_Toc124100036)

[**Figure 5.1:** Accuracy Graph of Trial 5 26](#_Toc124100037)

[**Figure 5.2:** Error Rate Graph of Trial 5 27](#_Toc124100038)

[**Figure 5.3:** Precision Graph of Trial 5 27](#_Toc124100039)

[**Figure 5.4:** Recall Graph of Trial 5 28](#_Toc124100040)

[**Figure 5.5:** Loss Graph of Trial 5 28](#_Toc124100041)

[**Figure 5.6:** Accuracy Graph of Trial 18 30](#_Toc124100042)

[**Figure 5.7:** Error Rate Graph of Trial 18 30](#_Toc124100043)

[**Figure 5.8:** Precision Graph of Trial 18 31](#_Toc124100044)

[**Figure 5.9:** Recall Graph of Trial 18 31](#_Toc124100045)

[**Figure 5.10:** Loss Graph of Trial 18 32](#_Toc124100046)

[**Figure 5.11:** Overall Final Accuracy Graph of All Experiments 32](#_Toc124100047)

[**Figure 5.12:** Overall Final Error Rate Graph of All Experiments 33](#_Toc124100048)

[**Figure 5.13:** Overall Final Precision Graph of All Experiments 33](#_Toc124100049)

[**Figure 5.14:** Overall Final Recall Graph of All Experiments 34](#_Toc124100050)

List of Abbreviations

|  |  |
| --- | --- |
| CPU  CNN | Central Processing Unit  Convolutional Neural Network |
|  |  |
| FLAIR | Fluid-Attenuated Inversion Recovery |
|  |  |
| GPU | Graphics Processing Unit |
|  |  |
| MRI | Magnetic Resonance Imaging |
|  |  |
| ReLU | Rectified Linear Unit |

List of Contents

[Authorization Form i](#_Toc124100051)

[Committee Decision i](#_Toc124100052)

[Dedication ii](#_Toc124100053)

[Acknowledgments iii](#_Toc124100054)

[List of Tables iv](#_Toc124100055)

[List of Figures v](#_Toc124100056)

[List of Abbreviations vii](#_Toc124100057)

[List of Contents viii](#_Toc124100058)

[Abstract ix](#_Toc124100059)

[Chapter 1 Introduction 1](#_Toc124100060)

[Chapter 2 Information About the Dataset 2](#_Toc124100061)

[Chapter 3 Methodology Used 4](#_Toc124100062)

[Chapter 4 Implementation 5](#_Toc124100063)

[Chapter 5 Dataset Experiments 24](#_Toc124100064)

[1.1 Introduction 24](#_Toc124100065)

[1.2 Dataset Experiments 1 24](#_Toc124100066)

[1.3 Dataset Experiments 2 28](#_Toc124100067)

[1.4 Comparing the Best Results 32](#_Toc124100068)

[1.5 Overall Results Comparison 32](#_Toc124100069)

[Chapter 6 Final Results 35](#_Toc124100070)

[6.1 Conclusions 35](#_Toc124100071)

[6.2 Limitations 35](#_Toc124100072)

[References x](#_Toc124100073)i

Brain Tumor MRI Binary Classification Study

**By**

**Hussein Kasim**

**Supervisor**

**Samer Nofal**

Abstract

Brain tumors are a significant health issue, with malignant tumors being a leading cause of cancer-related deaths. Accurate diagnosis is crucial for the effective treatment of patients, and Magnetic Resonance Imaging (MRI) is a commonly used tool in the evaluation of brain tumors. In this study, a binary classifier was used to differentiate between cases of benign and malignant brain tumors using MRI scans.

To train and test our classifier, a dataset of fluid-attenuated inversion recovery (FLAIR) MRI images and labels was found online and split into training and validation datasets. The scans were processed, and features were extracted, and these features were used to train a Convolutional Neural Network (CNN) machine learning model based on the ResNet architecture to classify the tumors as benign or malignant. The model’s performance was then evaluated and tested via the validation dataset twice, each time using a different percentage split of the entire dataset for training and validation. A total of twenty-four experiments were carried out, each with varying hyperparameters to observe the change in performance. The results of these experiments were recorded and analyzed.

This study was done using the Python programming language on the PyTorch machine learning framework. Many modules were used in unison with PyTorch, including NumPy, scikit-learn, and Matplotlib being some of the more prominent modules used. The Google Colab environment was also used due to the availability of Graphics Processing Units (GPUs), which were vital for the computations and processing of the images in the dataset.

The results of this study showed that it is feasible to use MRI scans to differentiate between benign and malignant brain tumors, but that they cannot be entirely relied on. In the most accurate experiment, the classifier achieved a result of 97.6% accuracy on the validation dataset, indicating that it could be a useful tool for improving the diagnosis and treatment of brain tumors, to a certain extent. Further research is needed to validate and optimize the classifier, but these initial results are quite promising.

**Keywords:** Brain tumors, Magnetic Resonance Imaging (MRI), binary classifier, Convolutional Neural Network (CNN), ResNet machine learning model, dataset, experiments, Python, PyTorch, NumPy, scikit-learn, Matplotlib, Google Colab, Graphics Processing Unit (GPU).

# Chapter 1 Introduction

Medical imaging plays a vital role in the diagnosis and treatment of various diseases. It allows healthcare professionals to visualize the internal structure and function of the human body and enables them to identify abnormalities and make informed decisions about patient care. With the advancement of technology, medical imaging modalities, such as computed tomography (CT) and magnetic resonance imaging (MRI) have become increasingly sophisticated and capable of producing high-resolution images.

However, the interpretation of medical images can be challenging and time-consuming, even for experienced radiologists. This is where deep learning, a branch of artificial intelligence, comes into the picture. Deep learning algorithms can learn from large amounts of data and recognize patterns and features that may not be apparent to the human eye. By detecting these patterns and features, classification models, such as the ones covered in this thesis, can classify the images given to them into different classes.

The field of medical imaging is one for the future and is a stepping-stone to accurate and automated healthcare services being made available to all. Thus, it is a field in need of as much research and advancement as possible and I wish that this thesis can act as inspiration to others to put effort into developing this ever-growing field.

In this thesis, we will use a fluid-attenuated inversion recovery (FLAIR) brain tumor MRI dataset applied to a ResNet Deep Learning model and discuss the results of various binary classification experiments on said dataset.

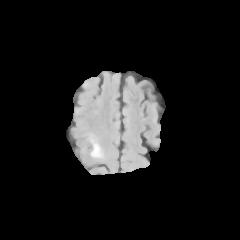
# Chapter 2 Information About the Dataset

The dataset used was a brain tumor MRI dataset that was published publicly on Kaggle.com [1]. This dataset includes 3,762 images of fluid-attenuated inversion recovery (FLAIR) brain tumor MRI scans. This number includes 1,683 images containing tumors and 2,079 images not containing tumors. FLAIR is a technique used to brighten areas of tissue while darkening bones and cerebrospinal fluid, thereby exposing lesions more easily.

This dataset was selected for multiple reasons:

* It was readily available.
* It included a fair number of images.
* Fluid-attenuated inversion recovery (FLAIR) was applied to ease pattern detection.
* The images included were all cropped to a 240x240 pixel size, producing a consistent image size that is easier to work with.

An example image from the dataset:



**Figure 2.1:** An Example Image

To use this dataset for this study, it was required to split the dataset into two smaller datasets, called the training and validation datasets. Since two experiments, were carried out, the image split between the training and validation datasets was done twice. The validation dataset was also used as a testing dataset at the end.

For one of these experiments, a 90-10 split was used, so that 90% of the images were randomly selected for the training dataset and 10% of the images were randomly selected for the validation dataset. This resulted in 3,386 images being selected for training and 376 being selected for validation. The validation dataset was also used for testing.

As for the other experiment, an 80-20 split was used, so that 80% of the images were randomly selected for the training dataset and 20% of the images were randomly selected for the validation dataset. This resulted in 3,010 images being selected for training and 752 being selected for validation. The validation dataset was also used for testing.

Since the total number of images was quite low, data augmentation and normalization techniques, such as random cropping and random horizontal flips, were used to vary the images to produce a model that can generalize accurately. (These will be explained in greater detail in Chapter 4).

# Chapter 3 Methodology Used

The programming language used to implement this model was Python, along with the Machine Learning framework PyTorch and numerous other modules, such as:

* os
* torch
* torch.nn
* torch.nn.functional
* torch.utils.data
* numpy
* torchvision.transforms
* torchvision.utils
* torchvision.datasets
* matplotlib.pyplot
* google.colab
* zipfile
* sklearn.metrics

The study was also conducted on Google Colab, as a Graphics Processing Unit (GPU) was required to train the images and GPUs are readily available on Google Colab.

Each of the above modules provided a unique functionality to complete this study, as explained below:

* The os module was used to interact with the operating system and access directories.
* The torch module is the module which implements the functions of the PyTorch framework.
* The torch.nn module provided the basic functions to build a neural network. (With alias set to nn).
* The torch.nn.functional module provided further functions to build a neural network, such as the cross-entropy loss function. (With alias set to F).
* The torch.utils.data module was used to import the Dataloader function.
* The numpy module was used to work with NumPy arrays and perform mathematical operations on these arrays. (With alias set to np).
* The torchvision.transforms module goes hand-in-hand with PyTorch and torch and was used to perform image transformations on the dataset. (With alias set to tt).
* The torchvision.utils module goes hand-in-hand with PyTorch and torch and was used to display some images in the dataset as a grid for the purpose of this thesis.
* The torchvision.utils module goes hand-in-hand with PyTorch and torch and was used to manipulate the extracted folder containing the images, in order to split them into different datasets (training and validation).
* The matplotlib.pyplot module was used to create visualizations, such as graphs, that represent the results of the study. (With alias set to plt).
* The google.colab module was used to mount Google Drive into Google Colab and allowed access to files on Google Drive.
* The zipfile module was used to unzip files imported from Google Drive.
* The sklearn.metrics module was used to implement metric calculations, such as accuracy score, error rate score, precision score, and recall score on the results of each epoch.

The process followed involved multiple steps. The images were first imported to the Google Colab notebook from Google Drive as a zip file. The images were then extracted into the local directory. Next, data augmentation and normalization transformations were applied. These included a random crop transformation that would choose random images to crop and a random horizontal flip transformation that would choose random images to flip horizontally. These transformations were applied to introduce image variety into the dataset. Next, the data was split into a training dataset and a validation dataset. Afterwards, the new datasets were loaded onto the GPU. The ResNet model was then initialized and loaded onto the GPU as well.

The training dataset was then sent into the training loop in fixed batches for an experiment to be carried out; a batch value that would be changed between experiments. The validation dataset followed by being passed into the evaluation function, also in fixed batches, to evaluate metrics after the training phase had completed for each batch. The mean of each of these metric results was then calculated to produce the value of each metric per epoch. This process was repeated for a fixed number of epochs for an experiment; an epoch value that was also changed between experiments. The values of the final epoch represented the results of each experiment. Twenty-four total experiments were done and these will be explained in greater detail in Chapter 5.

The loss function used in this study was the cross-entropy loss function and the optimizer used was Adam.

The ResNet model used was a convolutional neural network (CNN) that is based on the ResNet architecture. The ResNet architecture was designed to address the problem of vanishing gradients in deep CNNs, which can make training difficult. The ResNet architecture introduces skip connections, which allow the gradients to bypass certain layers and flow more easily through the network.

The ResNet model used was described in a blog series published by Myrtle.ai [2] and consisted of two convolutional blocks, followed by a residual block, repeated twice. The original input value was added to the output of each of the residual blocks.

Each convolutional block contained a weight layer (convolutional layer), batch normalization layer, and a Rectified Linear Unit (ReLU) activation layer.

Each residual block consisted of two convolutional blocks. The residual blocks had the original input value added to their output, as well.

Graphical user interface, diagram

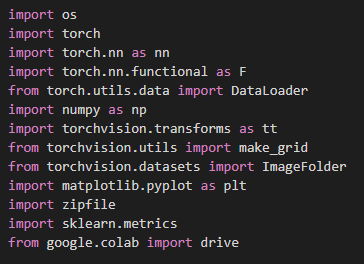
Description automatically generated

**Figure 3.1:** The ResNet Architecture Used

# Chapter 4 Implementation

This chapter covers the implementation of the model. This implementation follows a similar implementation of a similar project by Aakash N S [3]. Any function, variable, parameter, or decorator name used in the implementation is surrounded by single quotes (‘ ‘) when referred to in the explanations in this chapter for readability purposes.

At first, the modules used were imported: (The usage of each module was explained in Chapter 3.)



**Figure 4.1:** Imported Modules

Next, the Google Drive of the connected Google account was mounted using the ‘mount’ function of the google.colab module’s drive function. Afterwards, the zipfile module’s ZipFile function was used to unzip the zip file imported from Google Drive that contained the entire dataset. (The dataset was uploaded to Google Drive manually prior to the experiment). The dataset shown in the figure below is the dataset with the 90-10 data split between the training and validation datasets respectively. This dataset was used for the first twelve experiments and the 80-20 data split between the training and validation datasets respectively was used for the final twelve experiments.

Text

Description automatically generated

**Figure 4.2:** Importing the Dataset

Next, the directories were checked. The images were already randomly split manually into two directories ‘train’ and ‘validation’. Each of these two directories included a ‘0’ folder, which included non-tumor images and a ‘1’ folder, which included tumor images.

Text

Description automatically generated

**Figure 4.3:** Checking the Directories

A batch size was initialized and set to 16. This was used for the data normalization, which requires an initial batch size. To apply normalization techniques, the data was transformed into PyTorch tensors (using the ‘Compose’ function to apply transformations, which, in this case is only ‘ToTensor’, to convert the data into tensors). The two datasets, training and validation, were loaded onto the Central Processing Unit (CPU) using the ‘ImageFolder’ function and were split into ‘train\_ds’ and ‘valid\_ds’ respectively and had the transformation applied to them. The datasets ‘train\_ds’ and ‘valid\_ds’ were then loaded onto the dataloaders ‘train\_dl’ and ‘valid\_dl’ respectively.

The ‘num\_workers’ parameter specified how many processes were working in parallel to load the data and the ‘pin\_memory’ parameter was set to True to prepare the data to be loaded onto a GPU later.

Text

Description automatically generated

**Figure 4.4:** Loading the Data

Next, the mean and standard deviation were calculated to act as the data normalization function parameter. The following function was published in an article by Binary Study. [4]

Two empty tensors were created (‘fst\_moment’ and ‘snd\_moment’). By looping through all the images in the dataloader, the function calculated the number of pixels by multiplying the batch size by the height and by the width of the images (retrieved from the ‘images.shape’ function). The empty tensors ‘fst\_moment’ and ‘snd\_moment’ were then updated with the calculations of the mean and second moment (variance) respectively. The ‘cnt’ variable was then updated with the number of pixels of the current image before moving to the next.

The variance was calculated again using the equation: ‘(snd\_moment – fst\_moment \*\* 2)’. This variance equation produces a variance value with better statistical properties than the second moment variance value. The variance equation calculates the average of the squared differences from the mean, compared to the second moment, which calculates the average of the squared values of the data. Using this variance equation ensured that the data was properly normalized.

The standard deviation was calculated by taking the square root of the variance value.

The training dataset mean, training dataset standard deviation, validation dataset mean, and validation dataset standard deviation were all calculated.

Text

Description automatically generated

**Figure 4.5:** Calculating the Mean and Standard Deviation

In the first twelve experiments, the mean and standard deviation values calculated differed than the mean and standard deviation values of the last twelve experiments, as the data split was different. The results of each are shown below:

Text

Description automatically generated

**Figure 4.6:** Experiments 1-12 Mean and Standard Deviation Values

Text

Description automatically generated

**Figure 4.7:** Experiments 13-24 Mean and Standard Deviation Values

These values were stored as tuples in the ‘train\_stats’ and ‘valid\_stats’ variables for the training dataset values and validation dataset values. Transformations were applied to improve the diversity of the dataset. This improves the generalizability of the dataset, as it is considered too small for training, thereby producing a trained model capable of classifying a wide variety of image types. This process is called data augmentation and was applied to the data.

The transformations applied were:

* Random cropping
* Random horizontal flipping
* Normalization

For normalization, a required parameter is a tuple including the mean and standard deviation of the dataset, which were calculated and stored in ‘train\_stats’ for the training dataset and ‘valid\_stats’ for the validation dataset.

These transforms were then applied to the training and validation datasets using the ‘ImageFolder’ function and stored in ‘train\_ds’ and ‘valid\_ds’ respectively.

Text

Description automatically generated

**Figure 4.8:** Data Augmentation and Normalization

The batch size was then initialized to 16 or 32 depending on the experiment. The ‘train\_ds’ and ‘valid\_ds’ datasets (with data augmentation and normalization applied) were then loaded into the CPU.

Text

Description automatically generated

**Figure 4.9:** Data Loaded Into the CPU

The following functions and class moved the training and validation dataloaders ‘train\_dl’ and ‘valid\_dl’ from the CPU to the GPU. The CUDA device referred to the GPU.

The ‘get\_default\_device’ function selects the GPU if it is available and selects the CPU if the GPU is not available.

The ‘to\_device’ function moves a tensor to the device.

The ‘DeviceDataLoader’ class is used to move a dataloader to a new device.

The device was set to the GPU as Google Colab allowed the usage of a GPU. This was the main reason that Google Colab was chosen as the environment of this study.

Text

Description automatically generated

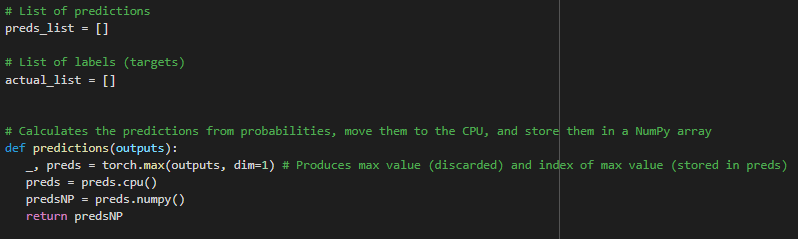
**Figure 4.10:** Device Dataloader

Both the ‘train\_dl’ and ‘valid\_dl’ dataloaders were then moved to the new device (GPU).



**Figure 4.11:** Dataloaders Moved to the GPU

Two empty lists were created and were later filled by the model predictions and labels (targets) of the dataset. A function was defined that took in the raw, unnormalized predicted probabilities and generated the predictions from them, by taking the maximum value’s index, which represented the class, and stored it in the ‘preds’ variable.



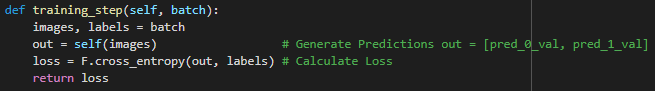
**Figure 4.12:** Predictions Function

An image classification base class (‘ImageClassificationBase’) was then created, and extended the base nn.Module class provided by torch.nn, in order to add neural network functionality. This class is comprised of four functions, which perform necessary tasks for the training loop when it is run. The four functions are:

* ‘training\_step’: Performs the training step on a batch of data.
* ‘validation\_step’: Performs the validation step on a batch of data.
* ‘validation\_epoch\_end’: Calculates the metrics per epoch.
* ‘epoch\_end’: Prints the metric results in a formatted form.

These functions will be explained one at a time, starting with the ‘training\_step’ function:

This function takes in one batch of data and extracted the images and labels. The images are then passed into the model’s architecture (which will be explained in greater detail later in this chapter) to generate predictions. A cross-entropy loss function follows and produces a loss value by comparing the generated prediction and the label of each image in the given batch. This loss value is returned. (The usage of this loss value will be explained in greater detail later in this chapter).



**Figure 4.13:** Training Step Function

The ‘validation\_step’ function performs the same functionality as the training function, in addition to passing the generated predictions to the aforementioned ‘predictions’ function and returning the class predictions for each image in the given batch. Also, the aforementioned ‘preds\_list’ list is filled with all the predicted class values and the ‘actual\_list’ list is filled with all the label values of the images in the batch. The loss value is returned to the CPU.

A screenshot of a computer

Description automatically generated with medium confidence

**Figure 4.14:** Validation Step Function

The ‘validation\_epoch\_end’ is used to calculate the metrics per epoch. The rest of the metrics are calculated by using functions from the sklearn.metrics module. These metrics include accuracy score, error rate score, precision score, recall score, and the confusion matrix. The meanings of these scores are as follows:

* Accuracy Score: The percentage of how many images were correctly predicted.
* Error Rate Score: The percentage of how many images were incorrectly predicted.
* Precision Score: Quantifies the quality of a positive prediction made by the model. (Correctly predicted positive cases divided by the total predicted positive cases).
* Recall Score: Quantifies the number of correctly identified positive cases. (Correctly predicted positive cases divided by the total actual positive cases).
* Confusion Matrix: A matrix displaying true positive, true negative, false positive, and false negative values.

A positive case refers to the presence of a tumor and a negative case refers to no tumor being present.

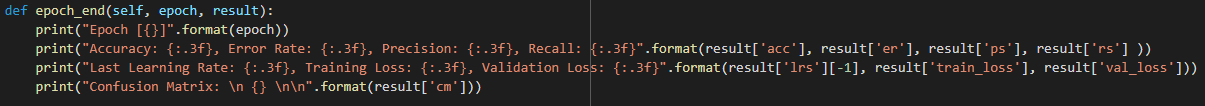
The epoch loss metric is the only metric that did not require a sklearn function as it is calculated in the ‘validation\_step’ function. Using the returned ‘val\_loss’ value, which represents the validation loss of one batch, the epoch validation loss can be calculated by taking the mean of the ‘val\_loss’ values of all batches. The ‘preds\_list’ and ‘actual\_list’ lists are also cleared between epochs.

Graphical user interface, text

Description automatically generated

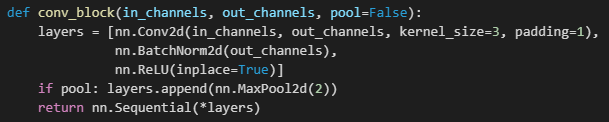
**Figure 4.15:** Validation Epoch End Step Function

The ‘epoch\_end’ function will print all the calculated metric values in a readable format. This function marks the end of the ‘ImageClassificationBase’ class.



**Figure 4.16:** Epoch End Function

A ‘conv\_block’ function was defined to pass the datasets into a convolutional block. When calling this function, the input channels and required output channels for the convolutional layer are passed in as parameters, along with an optional ‘pool’ parameter that, when set to True, includes a max pooling layer at the end of the block. This function creates a convolutional block comprising of a convolutional layer, a batch normalization layer and a ReLU activation layer. This function is used in the ResNet class, which will be explained later.



**Figure 4.17:** Convolutional Block Function

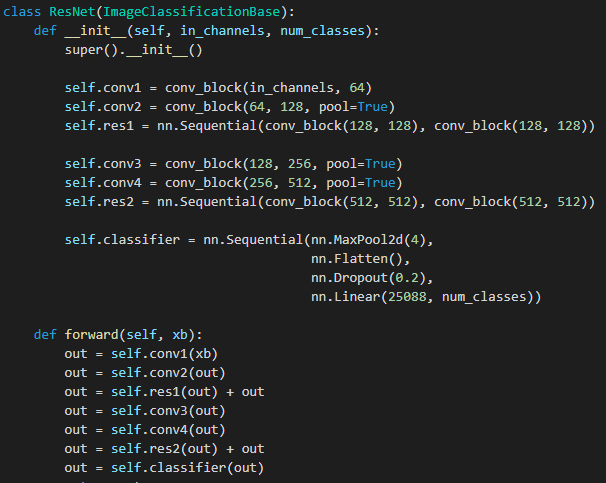
The ‘ResNet’ class extends the ‘ImageClassificationBase’ class and includes the ResNet architecture used. It is comprised of two convolutional blocks, followed by a residual block, which includes two convolutional blocks with the original input added back to the residual block’s outputs, repeated twice.

The class also contains a classifier block consisting of:

* A Max Pooling Layer: To apply max pooling with a kernel size of 4.
* A Flatten Layer: To reduce the tensor to a one-dimensional vector.
* A Dropout Layer: To set 20% of inputs to zero during training to reduce overfitting.
* A Fully Connected Layer: To perform linear transformation on the input.

The classifier block takes the output of the CNN and produces class probabilities as the output.

The ‘forward’ function performs the forward propagation of the model. It takes in the input data and passes it though the ResNet model. The final output (array of probabilities) is returned.



**Figure 4.18:** ResNet Class

The model is then moved to the GPU. 3 input channels (RGB) are passed in and 2 output channels (classes) are outputted.



**Figure 4.19:** Model Moved to the GPU

The training and validation loop were defined next to train, and update gradients of the model, and to evaluate the metrics respectively. The training loop used involved three special additions that were added to increase the model’s accuracy:

* Weight Decay: Prevents the weights from becoming too large by adding a penalty term to the loss function. This causes the optimizer to minimize the values of the parameters (weights). This reduces overfitting.
* Gradient Clipping: Rescaling gradients if they get too large to prevent undesirable changes in the model, such as exploding gradients.
* Learning Rate Scheduling: The process of adjusting the learning rate during training.

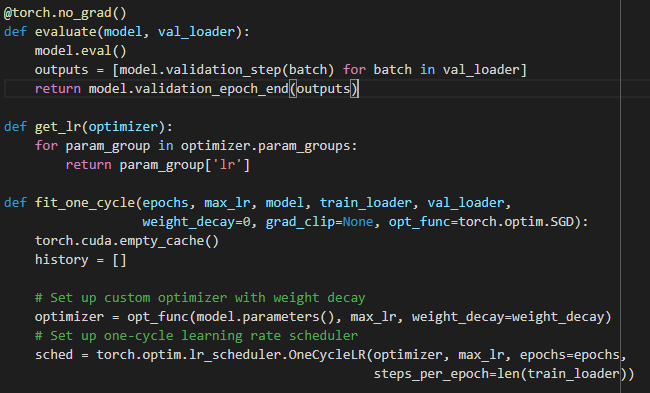
A ‘@torch.no\_grad()’ decorator is defined to indicate that the following function should not keep track of gradients. This function is the ‘evaluate’ function that evaluates the model on the validation dataset. The model is set to the ‘eval’ mode and then passes all the batches of data in the validation dataloader into the aforementioned ‘validation\_step’ function, the outputs of which are passed into the ‘validation\_epoch\_end’ function. The returned values are the calculated metrics.

The ‘get\_lr’ helper function takes in an optimizer and returns its current learning rate.

The ‘fit\_one\_cycle’ function acts as the training fit function using the one-cycle policy learning rate schedule, which starts with a high learning rate and decreases it over time. This leads to the model making rapid progress at the start of training and then fine-tuning the parameters near the end of training.

The GPU cache is cleared, and a ‘history’ list is created to record the results of the evaluation of the training and validation.

The optimizer is created, with parameters set for the maximum learning rate and weight decay values. A scheduler is also created and assigned to work on the one cycle policy.



**Figure 4.20:** Helper Functions and Variables Setup

The training loop code followed. It looped through each epoch and set the model to the train state. It then looped through every batch of data in the training loader and performed the training step, saving the losses in the ‘train\_losses’ list. The ‘backward’ function was called to compute the gradients of the loss. A check for a gradient clipping value was made, and if one existed, the values of the gradients are clipped between a minimum and maximum value (-1e-4 and +1e-4). The computed gradients were then used by the optimizer to update the model’s parameters (weights) in the next step in the ‘step’ function. The optimizer was then set to clear the calculated gradient values using the 'no\_grad’ function. The learning rate was also appended to the ‘lrs’ list and updated using the ‘step’ function.

The validation phase evaluated the model using the ‘evaluate’ function defined earlier and calculated the mean training loss. The results of the validation dataset evaluation were printed at the end of every epoch. The results were added to the ‘history’ list. This allowed us to later plot the results.

Text

Description automatically generated

**Figure 4.21:** Training and Validation Loop

An initial prediction was made and used as a starting point.



**Figure 4.22:** Initial Prediction

The hyperparameters were predefined and varied between experiments.

* Number of epochs: Set to either 10, 20, or 30 depending on the experiment.
* Learning Rate: Set to 0.01 or 0.05 depending on the experiment
* Gradient Clipping: Fixed at 0.1 (Time was not sufficient to vary this parameter across the study and to perform experiments).
* Weight Decay: Fixed at 1e-4 (Time was not sufficient to vary this parameter across the study and to perform experiments).
* Optimizer: Fixed to Adam (Time was not sufficient to vary this parameter across the study and to perform experiments).

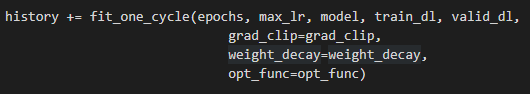
Although not mentioned here, as it was required for training, the batch size was also set to 16 or 32 depending on the experiment.

Text

Description automatically generated

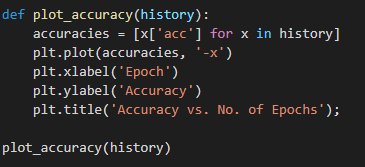
**Figure 4.23:** Hyperparameter Values

The ‘fit\_one\_cycle’ function was used to train and evaluate the model, while storing the results in history every epoch.



**Figure 4.24:** Fit Function

After obtaining the results, the accuracy score, error rate score, precision score, recall score and losses (training and validation) were all plotted.



**Figure 4.25:** Accuracy Plotting

Text

Description automatically generated

**Figure 4.26:** Error Rate Plotting

Text

Description automatically generated

**Figure 4.27:** Precision Plotting

Text

Description automatically generated

**Figure 4.28:** Recall Plotting

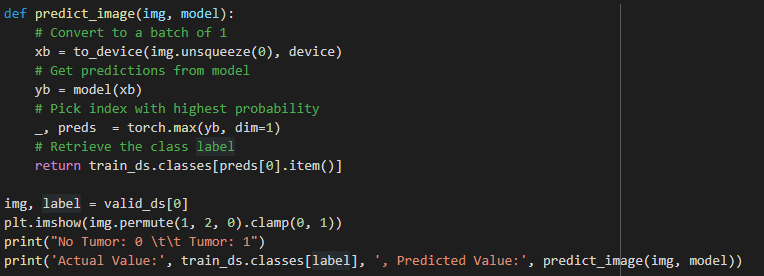
Text

Description automatically generated

**Figure 4.29:** Losses Plotting

For testing purposes, a ‘predict\_image’ function was created to pass a single given image into the model and produce a prediction for that image. The predicted value was returned.

The image was then selected (from the validation dataset) and displayed on a plot, along with its label value and predicted value.



**Figure 4.30:** Singular Image Testing on the First Image

An example of an image prediction where a tumor was correctly predicted:

A picture containing graphical user interface

Description automatically generated

**Figure 4.31:** Tumor Detected Example

An example of an image prediction where the lack of a tumor was correctly predicted:

Graphical user interface

Description automatically generated

**Figure 4.32:** No Tumor Detected Example

# Chapter 5 Dataset Experiments

## Introduction

As previously stated, twenty-four total experiments were conducted on the dataset. Twelve of these experiments were conducted with 90% of the initial dataset data placed used as training data and 10% as validation and testing data. The other twelve experiments were conducted with 80% of the initial dataset data placed used as training data and 20% as validation and testing data.

During the experiments, three factors were varied (besides the data split mentioned above):

* Number of epochs: 10, 20, or 30 epochs were used.
* Batch size: 16 or 32 batch sizes were used.
* Learning Rate: 0.01 or 0.05 were used.

Each of these values was tried with all the others, for each of the dataset data splits, resulting in 24 experiments (4!).

The following two sections of this chapter will consist of the analysis of each data split, followed by a section briefly comparing their best results. The final section will compare the overall results of both sets of experiments. Data Experiments 1 will cover the first twelve experiments with the 90-10 data split and Data Experiments 2 will cover the final twelve experiments with the 80-20 data split.

## Dataset Experiments 1

For these experiments, 90% of the initial dataset data placed used as training data and 10% as validation and testing data. A total of 3,386 images were used for the training data and a total of 376 images were used for the validation dataset. Out of the 3,386 images, 1,515 images contained a tumor, and 1,871 images did not contain a tumor. Out of the 376 validation images, 168 images contained a tumor and 208 did not contain a tumor. Some values were kept constant, such as:

* The optimizer used was fixed to only the Adam optimizer
* The Gradient Clipping value was fixed to 0.1
* The Weight Decay value was fixed to 1e-4.

The following table contains the resulting values. Each row represents a different trial:

**Table 5.1:** Experiments 1-12 Results

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Trial | Epoch | Batch Size | Learning Rate | Initial Accuracy | Accuracy | Error Rate | Precision | Recall | True Positive | True Negative | False Positive | False Negative |
| 1 | 10 | 16 | 0.01 | 0.553 | 0.963 | 0.037 | 0.994 | 0.923 | 207 | 155 | 1 | 13 |
| 2 | 10 | 32 | 0.01 | 0.489 | 0.976 | 0.024 | 0.982 | 0.964 | 205 | 162 | 3 | 6 |
| 3 | 20 | 16 | 0.01 | 0.41 | 0.973 | 0.027 | 0.988 | 0.952 | 206 | 160 | 2 | 8 |
| 4 | 20 | 32 | 0.01 | 0.62 | 0.971 | 0.029 | 0.988 | 0.946 | 206 | 159 | 2 | 9 |
| 5 | 30 | 16 | 0.01 | 0.447 | 0.976 | 0.024 | 0.988 | 0.958 | 206 | 161 | 2 | 7 |
| 6 | 30 | 32 | 0.01 | 0.447 | 0.968 | 0.032 | 0.981 | 0.946 | 205 | 159 | 3 | 9 |
| 7 | 10 | 16 | 0.05 | 0.447 | 0.891 | 0.109 | 0.899 | 0.851 | 192 | 143 | 16 | 25 |
| 8 | 10 | 32 | 0.05 | 0.513 | 0.96 | 0.04 | 0.994 | 0.917 | 207 | 154 | 1 | 14 |
| 9 | 20 | 16 | 0.05 | 0.606 | 0.894 | 0.106 | 0.905 | 0.851 | 193 | 143 | 15 | 25 |
| 10 | 20 | 32 | 0.05 | 0.559 | 0.931 | 0.069 | 0.913 | 0.935 | 193 | 157 | 15 | 11 |
| 11 | 30 | 16 | 0.05 | 0.449 | 0.96 | 0.04 | 0.987 | 0.923 | 206 | 155 | 2 | 13 |
| 12 | 30 | 32 | 0.05 | 0.548 | 0.92 | 0.08 | 0.937 | 0.881 | 198 | 148 | 10 | 20 |

For simplicity, only the best trials will be thoroughly discussed. Going by accuracy, the best trials were trial 2 and 5. Both reported an accuracy of 97.6% The main difference between these two trials was that trial 2 reported higher precision than trial 5; however, trial 5 reported higher recall than trial 2. This meant that trial 2 recorded more false negative cases than trial 5 and trial 5 recorded more false positive cases than trial 2. These two trials are extremely alike, and it is difficult to judge which of the two performs better. Since this is a medical imaging problem, our goal must be more focused on reducing false negative cases rather than false positive cases.

A false positive case means that a tumor was detected by the model, without the actual presence of one. In the real world, this might cause some panic and fear for the patient and must be avoided, but this always results in good patient’s well-being and health.

A false negative case; however, is more terrifying. It is a case where a tumor was not detected by the model, with the actual presence of one. In the real world, this results in the patient being diagnosed as healthy, when there is, in fact, a tumor present. These cases must be detected as soon as possible, to ensure that the patient receives proper treatment as quickly as possible. So, false negative cases are more important to detect than false positive cases as they may cause serious harm to the patients.

As such, the better model must return a lower false negative value, which is the case in trial 5. Trial 5’s parameters included: 30 epochs, a batch size of 16, and a learning rate of 0.01. This leads to the conclusion that when there is a 90-10 data split, it is best to use a larger number of epochs, a smaller batch size, and a lower learning rate.

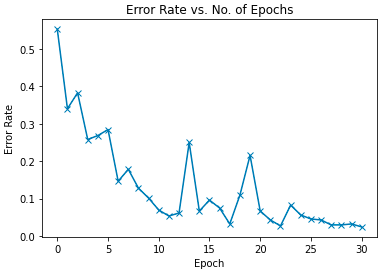
Trial 5’s accuracy graph displayed a consistent upward trend during training.

Chart, line chart

Description automatically generated

**Figure 5.1:** Accuracy Graph of Trial 5

Trial 5’s error rate graph is a mirror image of the accuracy graph.



**Figure 5.2:** Error Rate Graph of Trial 5

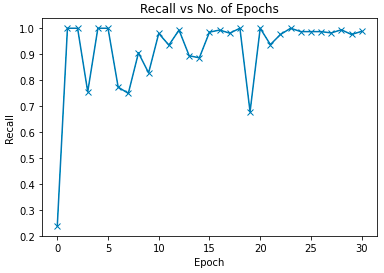
Trial 5’s precision graph also developed a consistent value towards the last 6 epochs, highlighting the importance of the last 6 epochs. Trial 5 showed high precision, meaning that the quality of positive cases was high.

Chart, line chart

Description automatically generated

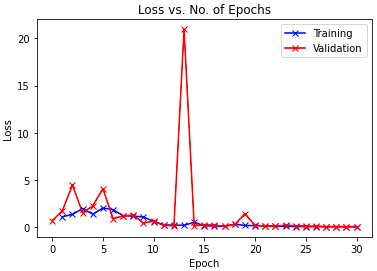
**Figure 5.3:** Precision Graph of Trial 5

Trial 5’s recall graph also developed a consistent value towards the last 10 epochs, highlighting the importance of the last 10 epochs. Trial 5 showed high recall, further proving the results of the precision graph; that the quality of positive cases was high.



**Figure 5.4:** Recall Graph of Trial 5

The loss graph showed that the training loss dropped and remained stable around 0.1, while the validation loss remained mostly low and remained stable around the same value as the training loss. These results indicate that an almost optimal fit was reached.



**Figure 5.5:** Loss Graph of Trial 5

## Dataset Experiments 2

For these experiments, 80% of the initial dataset data placed used as training data and 20% as validation and testing data. A total of 3,010 images were used for the training data and a total of 752 images were used for the validation dataset. Out of the 3,010 images 1,347 images contained a tumor and 1,663 images did not contain a tumor. Out of the 752 validation images, 336 images contained a tumor and 416 did not contain a tumor. Some values were kept constant, such as:

* The optimizer used was fixed to only the Adam optimizer
* The Gradient Clipping value was fixed to 0.1
* The Weight Decay value was fixed to 1e-4.

The following table contains the resulting values. Each row represents a different trial:

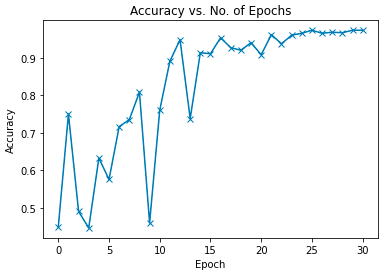
**Table 5.2:** Experiments 13-24 Results

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Trial | Epoch | Batch Size | Learning Rate | Initial Accuracy | Accuracy | Error Rate | Precision | Recall | True Positive | True Negative | False Positive | False Negative |
| 13 | 10 | 16 | 0.01 | 0.471 | 0.963 | 0.037 | 0.978 | 0.938 | 409 | 315 | 7 | 21 |
| 14 | 10 | 32 | 0.01 | 0.463 | 0.955 | 0.045 | 0.949 | 0.949 | 399 | 319 | 17 | 17 |
| 15 | 20 | 16 | 0.01 | 0.527 | 0.963 | 0.037 | 0.972 | 0.943 | 407 | 317 | 9 | 19 |
| 16 | 20 | 32 | 0.01 | 0.5 | 0.952 | 0.048 | 0.957 | 0.935 | 402 | 314 | 14 | 22 |
| 17 | 30 | 16 | 0.01 | 0.65 | 0.964 | 0.036 | 0.984 | 0.935 | 411 | 314 | 5 | 22 |
| 18 | 30 | 32 | 0.01 | 0.449 | 0.973 | 0.027 | 0.988 | 0.952 | 412 | 320 | 4 | 16 |
| 19 | 10 | 16 | 0.05 | 0.517 | 0.876 | 0.124 | 0.901 | 0.812 | 386 | 273 | 30 | 63 |
| 20 | 10 | 32 | 0.05 | 0.553 | 0.899 | 0.101 | 0.906 | 0.863 | 386 | 290 | 30 | 46 |
| 21 | 20 | 16 | 0.05 | 0.552 | 0.894 | 0.106 | 0.924 | 0.83 | 393 | 279 | 23 | 57 |
| 22 | 20 | 32 | 0.05 | 0.551 | 0.915 | 0.085 | 0.9 | 0.911 | 382 | 306 | 34 | 30 |
| 23 | 30 | 16 | 0.05 | 0.449 | 0.949 | 0.051 | 0.969 | 0.917 | 406 | 308 | 10 | 28 |
| 24 | 30 | 32 | 0.05 | 0.525 | 0.907 | 0.093 | 0.887 | 0.908 | 377 | 305 | 39 | 31 |

For simplicity, only the best trial will be thoroughly discussed. Going by accuracy, the best trials was trial 18. It reported an accuracy of 97.3%. It also showed the highest precision value and the highest recall value out of these twelve experiments, as expected. In addition, it showed the lowest false negative value of only four cases and the lowest false positive value of only 16.

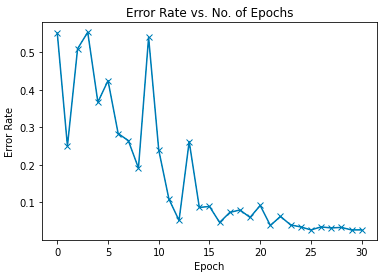
Trial 18’s parameters included: 30 epochs, a batch size of 32, and a learning rate of 0.01. This leads to the conclusion that when there is a 80-20 data split, it is best to use a larger number of epochs, a larger batch size, and a lower learning rate.

Trial 18’s accuracy graph displayed a consistent upward trend during training.



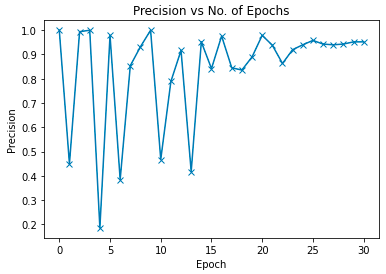
**Figure 5.6:** Accuracy Graph of Trial 18

Trial 18’s error rate graph is a mirror image of the accuracy graph.



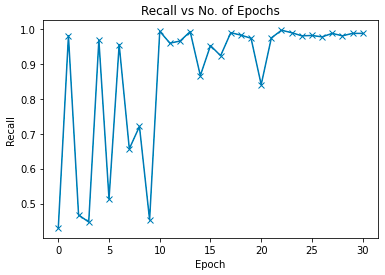
**Figure 5.7:** Error Rate Graph of Trial 18

Trial 18’s precision graph also developed a consistent value towards the last 7 epochs, highlighting the importance of the last 7 epochs. Trial 18 showed high precision, meaning that the quality of positive cases was high.



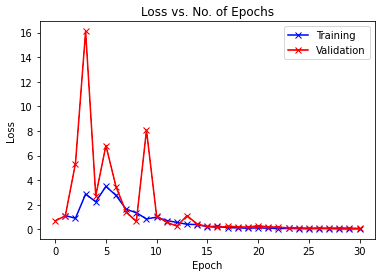
**Figure 5.8:** Precision Graph of Trial 18

Trial 18’s recall graph also developed a consistent value towards the last 9 epochs, highlighting the importance of the last 9 epochs. Trial 18 showed high recall, further proving the results of the precision graph; that the quality of positive cases was high.



**Figure 5.9:** Recall Graph of Trial 18

The loss graph showed that the training loss dropped and remained stable around 0.1, while the validation loss remained mostly low and remained stable around the same value as the training loss. These results indicate that an almost optimal fit was reached.



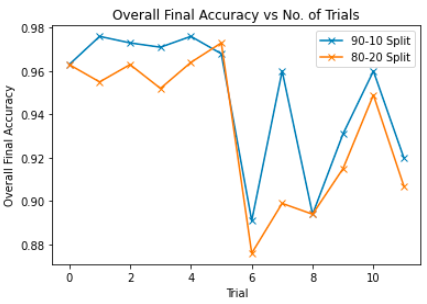
**Figure 5.10:** Loss Graph of Trial 18

## Comparing the Best Results

Overall, Trial 5 (97.6%) reported marginally better results than Trial 18 (97.3%). This proves that the best hyperparameters used included 30 epochs, a batch size of 16, a learning rate of 0.01 and a data split of 90% for training and 10% for validation. This also proves that more images being used for training benefit the model more than more images being used for validation.

## Overall Results Comparison

To further document these findings, graphs of the overall metrics across both sets of experiments are listed below:

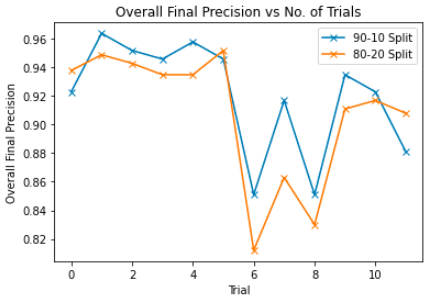


**Figure 5.11:** Overall Final Accuracy Graph of All Experiments

Chart, line chart

Description automatically generated

**Figure 5.12:** Overall Final Error Rate Graph of All Experiments



**Figure 5.13:** Overall Final Precision Graph of All Experiments

Chart, line chart

Description automatically generated

**Figure 5.14:** Overall Final Recall Graph of All Experiments

For the overall accuracy, overall error rate, and overall precision, both datasets follow a similar trend. There is only one instance among these three graphs where the 80-20 dataset performed better than the 90-10 dataset: error rate. This is a negative result. The recall graph is the only graph to look different for each dataset. Overall, the 80-20 split dataset reported an overall lower recall than the 90-10 split dataset.

These results, as well as the results of comparing the best trials from each set of experiments, prove that it is more beneficial to use more data to train the model, rather than using more data to evaluate the model.

# Chapter 6 Final Results

## 6.1 Conclusions

After producing promising results, some notable conclusions were reached:

The level of accuracy demonstrated by the ML model supports its use as a second reader or as part of a quality assurance system. Despite the great performance of this model, many false positive and false negative cases were recorded. These numbers would ideally be zero, giving a perfect model. That is unrealistic and so this model is unable to replace radiologists but might instead serve as a second opinion to them.

This model is further research on the topic of medical imaging and could prove useful in advancing the field, but more experiments must be conducted to reach more concrete results.

## 6.2 Limitations

Some limitations were also observed:

It was difficult to test the model’s generalizability without the use of a second dataset. No other datasets including FLAIR were available and so this could not have been done.

The size of the dataset used was also quite small for such a study (3,762 images). A larger dataset could not be provided.

The Weight Decay and Gradient Clipping values could have been experimented on, had there been more time given. The Adam optimizer could have also been changed to other optimizers and results could have provided an interesting insight to the efficiency of the different available optimizers on such a model. This could be considered a potential improvement for the current study for the future.

This study was not conducted on real world data, but instead on a dataset found online on kaggle.com. Real world data might provide different results, and it would be interesting to compare the results of different datasets created from MRI scans from all over the world. Efforts to receive real world data from local hospitals were not successful.

# References

|  |  |
| --- | --- |
| [1] | J. B. -. Kaggle, "Brain Tumor," [Online]. Available: https://www.kaggle.com/datasets/jakeshbohaju/brain-tumor. [Accessed 30 October 2022]. |
| [2] | Myrtle.ai, "How to Train Your ResNet," [Online]. Available: https://myrtle.ai/learn/how-to-train-your-resnet/. [Accessed 14 November 2022]. |
| [3] | A. N. S, "Classifying CIFAR10 images using ResNets, Regularization and Data Augmentation in PyTorch," [Online]. Available: https://jovian.ai/aakashns/05b-cifar10-resnet. [Accessed 31 October 2022]. |
| [4] | Binary Study, "How to Normalize Image Dataset in PyTorch," Binary Study, [Online]. Available: https://www.binarystudy.com/2022/04/how-to-normalize-image-dataset-inpytorch.html. [Accessed 24 December 2022]. |