CSDS312 Project: Final Report
for

Artificial Intelligence in Neuroradiology

The Multimodal Brain Tumor Segmentation Challenge

Eli Abboud, Ben Flock, Kevin Felt

ena12, bcf26, kcf26

May 11, 2021

Contents

1 Introduction

Recent advances in deep learning have made neural networks the methodology of choice for most perceptual problems. Unlike traditional statistics or alternative machine learning techniques, neural networks are uniquely well-suited to grapple with non-linear relationships and multi-modal data such as images, audio files, and videos. This technology has already shown promise for a variety of interdisciplinary applications. For instance, the United States Postal Service has leveraged neural networks to read hand-written mailing addresses and appropriately sort packages, and finance professionals have used them to gauge the popular sentiment around public companies via natural language processing. In a similar vein, our project seeks to apply deep learning to the field of neuroradiology. Specifically, by training a neural network on multimodal magnetic resonance images (MRIs) of the human brain, we hope to automate the process of brain tumor segmentation. If done effectively, this line of work can lead to improved outcomes for brain tumor patients and lower healthcare costs.

When a doctor suspects that a patient may suffer from a brain tumor, they are often referred to a neuroradiologist and subjected to a battery of medical tests, chief among them being MRI imaging. These MRI scans provide neuroradiologists with a detailed three-dimensional image of the patient's brain. Not only are these scans used to definitively determine a tumor's existence, but they also are used to assess a tumor's biological composition. Equipped with this information, a neuroradiologist and neurosurgeon can jointly categorize the tumor, determine the optimal treatment strategy, and provide the patient with a prognosis. Unfortunately, tumor structures can vary considerably across the dimensions of location, size, and shape. This heterogeneity can severely complicate the immensely important (and expensive) job of a neuroradiologist.

Fortunately, the University of Pennsylvania Perelman School of Medicine's Center for Biomedical Image Computing and Analytics has compiled a publicly available dataset of pre-operative MRI scans. This data, formally known as the Brain Tumor Image Segmentation Benchmark (BraTS), has subsequently been used by researchers to build machine learning models and essentially automate the job of neuroradiologists. More specifically, researchers have constructed supervised learning algorithms to denote the boundaries of each biological component of a tumor (i.e. edema, non-enhancing core, necrotic core, enhancing core).

For this project, our team intends to grapple with the BraTS dataset ourselves and implement our own neural network classifier. The architecture of our model is derived from an actual published work in this field [14]. This paper will proceed as follows. In Section II, we will provide some relevant medical background on brain tumors and their treatment. In Section III, we conduct exploratory data analysis upon the BraTS dataset. In Section IV, we will provide relevant background on artificial intelligence and explore the methods underlying our model. In Section V, we will discuss the implementation of our neural network. Finally, in Section VI, we will review our results and conclude.

2 Medical Background

Cancer is a broad descriptor for disease characterized by mutation-driven, uncontrolled cell proliferation. This uncontrolled cell growth eventually manifests as a mass of cells, referred to as a tumor. Tumors are named after the tissue and cell type from which they originate and have a range of pathologies. Tumors can be classified, generally, as benign or malignant, and are graded according to the World Health Organization (WHO) standards as grades I, II, III, or IV, ranking from least malignant (benign) to most malignant [2].

On the molecular level, the progression from healthy to cancerous tissue usually requires the accumulation of multiple driver mutations within a single cell population. Mutations can be acquired from birth (genetic predisposition) or from DNA-damaging sources such as ionizing radiation, oxidation (free radicals), or chemicals that have been shown to cause cancer (carcinogens). Genes that are commonly associated with cancer can usually be classified as either proto-oncogenes or tumor suppressors. Proto-oncogenes are genes that, once mutated, can acquire the ability to increase the rate of cell proliferation as compared to healthy cells (gain-of-function). Tumor suppressors are genes that help to prevent overproliferation in healthy tissues, and can no longer perform these tasks if mutated (loss-of-function) [1].

Brain tumors have a prevalence in the USA of 25 per 100,000 and have a high rate of mortality. Glioma are tumors that arise from the glial cells within the central nervous system and make up 70% of adult malignant primary brain tumors. Grade IV gliomas are termed glioblastoma multiforme and are characterized by rapid growth and invasion of healthy tissues, angiogenesis, edema and corresponding mass effect, and have a survival time of less than one year after diagnosis. Treatment for brain tumors usually consists of monitoring and imaging, surgery and tumor resection, radiation therapy, and chemotherapy [2].

2.1 MRI

The standard for brain tumor monitoring and imaging is Magnetic-Resonance Imaging (MRI). MRI is an important technique for glioma treatment because it is non-invasive, enabling a general diagnosis of the tumor without having to damage the patient. Briefly, MRI functions by manipulating the physical orientation of hydrogen atoms within the target through generation of a strong magnetic field and introduction of radiofrequency (RF) pulses. Hydrogen atoms, acting each as tiny magnets, will absorb energy from the magnetic field and RF pulses and align themselves accordingly. In absence of these energetic inputs, the hydrogen atoms will relax to their original positions at rates dependent on their local atomic environment, a process which can be detected by voltage sensors within the MRI machine. Each H atom in the 3D space of the target can produce a voltage signal, and the compilation of these signals is transformed into the MRI image [2][7].

H atoms can relax in space in two independent dimensions, each referred to as T1 and T2 relaxation. Each of these relaxations portrays a different image of the object, and MRI images can be weighted to show signal from either T1 or T2 relaxations. Common modes of MRI are T1 weighted, T1 weighted - contrast enhanced, T2 weighted, and T2-FLAIR. T1 weighted images show the general structure of the object, while T1-contrast enhanced images use an introduced indicator (usually gadolinium, injected into the vascular system of the patient) to reveal additional features with better contrast, such as the border of a tumor. T2 weighted images show edema around the tumor well, while T2-FLAIR images help distinguish between edema and cerebral spinal fluid. When all of these different modes are used to image a patient, it is termed multimodal-MRI (mMRI) [2][7].

2.2 Brain Tumor Segmentation

MRI images are usually annotated by highly-trained radiologists to determine the features of a brain tumor (also known as segmentation). However, researchers are now hoping to improve this diagnostic process by utilizing artificial intelligence (AI). Utilization of AI in glioma MRI segmentation could provide a more consistent diagnosis due to less human-generated error or variance. AI could also provide a better diagnosis of tumors from MRI by tracking image patterns and associations better than humans. Additionally, AI-driven diagnosis of MRI images for glioma could reduce medical costs due to alleviating the need for expensive diagnostic time from highly-trained humans. In accordance with these goals, the Multimodal Brain Tumor Image Segmentation Benchmark (BraTS) was established [11]. The BraTS competition established a standard MRI dataset and image analysis guidelines for automation of glioma segmentation. This benchmark allows creators of different algorithms for AI-driven glioma segmentation to directly compare their efforts to one another following the same common data and methodologies. Since the inception of BraTS, researchers have competed against one another to produce ever-more effective neural networks for classifying brain tumor MRI images.

3 Exploratory Data Analysis

The BraTS dataset consists of the MRI data of 352 patients, 285 of which are in the training partition and the remaining 67 are in the testing partition. Each patient is considered one independent observation and has a 4-dimensional 240x240x155x4 tensor associated with them. Two of the tensor's dimensions are used to enumerate MRI pixels. The third dimension is used to delineate between the 155 slices available in a single MRI. The final dimension is used to distinguish between the multi-modal MRI subscans (T1 MRI, T1ce MRI, T2 MRI, or FLAIR MRI).

Each multimodal MRI also has data labels associated with it. Much like the MRIs themselves, each label is a 240x240x155 tensor. However, the voxels are not represented by a hexadecimal value. Instead, the voxels are assigned a value between zero and four. A labelled value of zero indicates normal tissue; one indicates nonenhancing tumor tissue; two indicates an edema; three indicates necrotic tumor tissue; and four indicates enhancing tumor tissue. In total, the dataset is approximately 40GB large uncompressed and required the data transfer node to maneuver.

Before diving into the analysis, we attempted to familiarize ourselves with the dataset by conducting exploratory analysis. Using Python, Jupyter, Matplotlib, Pandas, and Numpy, we assembled a series of charts on the BraTS metadata. The first chart (Figure 1a) illustrated distribution between observations in the testing set and training set. The second (Figure 1b) showed the distribution of low and high-grade gliomas within the training set. The third (Figure 2a) depicted the average age of high-grade glioma patients. The final chart (Figure 2b) illustrated the residual lifespan of high-grade glioma patients.

Next, we explored the MRI images themselves. Using the NiLearn library, we were able to render the MRI slices (Figure 3). By leveraging NiWidgets, we were also able to create several interactive widgets for navigating the entire 3-dimensional MRI (Figures 4 and 5).[1]

4 Primer on Artificial Intelligence

For this project, we have employed a U-Net as our primary classifier. To fully appreciate the merits of this design choice, it is imperative to first have a deeper understanding of recent developments in the field of artificial intelligence. As such, this section will begin by reviewing the mechanics of artificial intelligence before navigating the breakthroughs that precipitated the invention of U-Nets.

4.1 Machine Learning

Since the onset of the computing age, software logic has been painstakingly implemented by human programmers. Video game designers account for every potential player decision when programming a gameplay; web developers must ensure that their site can respond to any possible user input; application developers must precisely code any and all functionality that they intend to offer. Furthermore, should a software engineer desire to customize their creation for particular users or end markets, they must build bespoke distributions for each possible environment.

Not only is explicitly defining all possible outcomes a cumbersome task, but it is also often a futile one. The most successful applications serve billions of users. For them, attempting to achieve user-level customization is financially infeasible. In addition, it is frequently unclear precisely how rules should be encoded. For instance, the difficulty in translating verbal language to programming logic has long stymied efforts at designing software capable of conversing.

The limitations inherent to this traditional paradigm of computer programming have motivated work in the young field of machine learning. Machine learning programs are able to deduce rules on their own. They are essentially able to learn by example. In practice, they are given millions of sample inputs as well as the desired output for each. Using a diverse statistical toolbox, these programs are able to internally tune an algorithm for future predictions on unlabelled data.

4.2 Neural Networks

Neural networks are one of the most popular machine learning techniques, particularly in the area of image recognition. Drawing inspiration from the human brain, they consist of a multitude of independent functions known as neurons. At a high level, a neural network learns in six steps:

1. Input Preprocessing: Machines only understand numbers. As such, a neural network architect must begin by numerically quantifying all of the data that they wish to train a neural network upon. For some datasets, this step is either superfluous or very

[1]While we obviously can not show off their interactivity in this document, we demoed these widgets during our presentation.

straightforward. For instance, a neural network designed to predict stock prices would likely be fed numeric data on historical stock prices and financial metrics. However, for perceptual data, this process is more of a challenge. For audio datasets, for example, the data must be parsed into a tensor of frequency, amplitude, and time. Similarly, for image data, the information is generally translated into a matrix of hexadecimal color values. The latter is actually the approach taken in this project.

2. Hidden Layer: A copy of the set of inputs are passed to each neuron in the neural network. Each of these neurons randomly weights and sums together the input values. Next, each neuron feeds the sum product into a nonlinear activation function. Finally, the neuron submits the result of its activation function to the output layer.

3. Output Layer: The output layer consists of one neuron for each outcome variable in the model. After receiving the hidden layer's results, the neuron again randomly generates an array of weights, calculates the sum product of all of the incoming data, and feeds the result into another activation function. This time, however, the result of the activation function corresponds to a potential outcome variable. For example, the output layer of our model returns an integer between zero and four, each corresponding to a different type of brain tissue.

4. Loss Function: The neural network then grades its performance. Using data labels (i.e. an answer key), the neural network assesses the accuracy of its predictions. It outputs a series of scores known as loss values.

5. Optimizer: The optimizer inspects the relationship between the randomly chosen weights and the loss values. Using an algorithm known as gradient descent, it estimates a new set of weights that would reduce the loss value and, by extension, improve the neural network's accuracy. The optimizer then updates the neurons in order to replace the old weights with the new, improved weights.

6. New Epoch: The neural network then redoes all of its calculations with the new batch of weights. Since the new weighting scheme is ideally more accurate, the loss value of this next round of calculations should be lower. This cycle is repeated until the loss values stabilize.

While this traditional neural network structure produces astonishing results in its own right, later work has shown that the performance of neural networks can be enhanced by including additional hidden layers such as in Figure 7 [3, 9, 10, 8]. Today, these "deep neural networks" are the tool of choice for countless applications.

4.3 Convolutional Neural Networks

However, neural networks have an important Achille's heel. Since each pixel is independently inputted into the network, the model has no concept of context or adjacency. In other words, the neural network is unable to understand which pixels neighbor each other. This is an especially acute problem in the field of computer vision because individual pixels rarely capture the distinguishing features of an image. Instead, only patches of adjacent pixels allow for proper classification. For instance, a neural network tasked with classifying different types of trees would likely perform best if it is able to process an entire leaf at once.

Convolutional neural networks (CNN) are the answer to this dilemma. They introduce two mathematical operations for aggregating information across multiple pixels: pooling and convolutions. A pooling function (Figure 8) accepts several hexadecimal values as inputs, performs an arithmetic operation upon them, and outputs a single result. The specific arithmetic operation varies between implementations, but sum, average, and maximum operations are common choices. Similarly, a convolution function (Figure 9) accepts several hexadecimal values as inputs, multiplies those inputs by an arbitrary matrix (known as a kernel or filter), and outputs a single result [4].

4.4 U-Nets

While convolution and pooling operations enhance the predictiveness of neural networks, these operations also transform the dimensions of the original inputs. As can be seen in Figures 8 and 9, the output tensor of a CNN layer is generally more compact than the input tensor. Although classification tasks can be carried out without issue, this reduction in dimensionality complicates localization tasks. In such tasks, the network architect intends to classify the individual pixels of the original image. In our case, for example, the network is supposed to label every pixel of the brain as normal tissue, nonenhancing tumor tissue, necrotic tumor tissue, enhancing tumor tissue, or an edema. Since the convolutional or pooling layers shrink and otherwise distort the dimensions of the original image, this task is infeasible with a traditional convolutional neural network.

U-Nets were invented as a method for retaining the enhanced predictiveness of convolutional neural networks while still allowing for localization tasks to be performed. They do this by partitioning the training process into two stages (Figure 10). In the first stage, the model is trained in an identical fashion as a traditional CNN. In the second stage, a process known as upsampling occurs. During upsampling, the hidden layers gradually reincorporate the data from earlier in the training process, recovering the dimensionality of the input data while retaining the model's predictiveness. By the end of this process, the dimensions of a neural network's output layer are identical to that of its input layer [16].

Since brain tumor classification requires both effective localization and stellar accuracy, a U-Net architecture was utilized in this project. The exact design is shown in Figure 11.

5 Implementation

In order to do a deep dive into the functionality and utility of the high performance computing cluster, it was necessary to find a real world application of these tools. Our goal in this project was to see how we could leverage the high performance computing cluster to implement and improve upon an existing model. We took advantage of a number of the built-in technologies of the HPC cluster in order to create an appropriate runtime environment for the U-Net model we had pulled from the Noori repository.

5.1 Slurm

Slurm is a job scheduling and resource management system used on HPC clusters. Slurm is responsible for ensuring that jobs receive the resources they need in order to complete. When submitting jobs, end users specify the requirements, and slurm will allocate those resources until the job completes, or until the job has reached the walltime limit[2]. Slurm tries to make full use of the resources available on the cluster. Multiple jobs are allowed to run on the same node, if there are enough resources to share. At the same time, jobs can request exclusive access to a node, even if it doesn't require all of the resources on the node.

Users are able to run interactive jobs with `srun`, and detached jobs with `sbatch`. A job initiated with `srun` will, in most cases, terminate when the connection is lost; on the other hand, a job that is run using `sbatch` is able to run on a node without a terminal session attached. [17]

Because the U-Net is such a sophisticated model, it became apparent that there were several features provided by Slurm that we would have to take advantage of. Since we were training a large model using tensorflow, we requested a single node with a single GPU.[3] [14] Another reason we used slurm was because of the runtime. It simply wasn't realistic to monitor a terminal session for the entire duration of the training process (See section 6).

5.2 LMOD Module System

One of the key features of the High Performance Computing environment was the Lmod module system. Lmod is one of a number of HPC tools that eases the process of creating a runtime environment. Lmod's website describes functionality of the tool as "a convenient way to dynamically change the users' environment through modulefiles", including "easily adding or removing directories to the `PATH` environment variable". This really undersells the power of the tool, as modifying `PATH` is a one-liner in bash or powershell. Where lmod excels is in dependency management. Lmod isn't a script to add binaries to the `PATH` variable. Lmod locates "Modulefiles", from the environment variable `MODULEFILES`, and uses these to build a repository of modules. The user is then able to load these modules into their HPC environment. Lmod will take care of dependency management. A properly written modulefile will be configured to add the module's dependencies to the path variable. When a module is unloaded, the binary is removed from the path. Lmod is able to run quickly because it doesn't need to retrieve the data. The binaries and libraries are already on the system; they just need to be added to the `PATH` variable.

5.2.1 Singularity

One of the modules we took advantage of was Singularity. Singularity is a container runtime, similar to Docker. The Docker container runtime is designed as a devops tool; it allows developers to concentrate more on development, spending less time on debugging configuration

[2]Walltime refers to the elapsed real-world time, as opposed to "cpu time", which is a total of the elapsed time measured by each processor. This is an important distinction in the realm of parallel computing

[3]We experimented with adding more nodes and more gpus; however, the model was designed for a single-node, single-gpu environment, and we didn't want to make those kinds of overhauls to the model.

problems and more time working on features. Specifically, Docker is one of the primary tools used in the Continuous Integration/Continuous Delivery pipeline, alongside Kubernetes, the primary container orchestration tool for cloud computing.

When a user installs docker, they're installing an entire tech stack. It used to be that most of the functionality was packaged in the docker engine, dockerd. In 2015, two years after release, the Docker team separated the docker daemon from the container runtime. Containerd works as a high level API for the functionality of the lower level runc, the daemon responsible for running and managing containers. This runc is not a Docker technology – rather, it was developed by the Open Containers Initiative. Docker images and containers are designed to an OCI specification.[5]

Singularity is another OCI compliant container runtime. Singularity is designed for scientific computing applications, rather than devops, making it one of the first, if not the first, meaningful development in container technology branching off from the Docker model. Docker isolates its container environment from the host environment, making it especially useful for ephemeral microservices. The developer is able to mount specific folders and files into the container environment. Singularity, on the other hand, directly mounts onto the host filesystem, and is able to make direct edits. In this way, Singularity more directly depicts the difference between virtualization and containerization.

The primary reason for using Singularity over other container runtimes on HPC environments is security. Docker requires root access to build images, whereas Singularity does not. As of Singularity 3.5, Singularity offers a powerful "fakeroot" feature, where the user has root privileges inside the scope of the container. This allows them to create and edit files as if they were root inside the container, as well as open container ports – which is a research paper in and of itself – while not having the permissions to modify files for which they would not have access to outside of the container. Singularity also offers a "sandbox" feature option when building images, which allows users to build images as folders and open a shell inside the container, from which they may make permanent changes to the image and later repackage it. This is incredibly useful for fine-tuning a container to the specific needs of a project.[18]

The Case Western HPC cluster has multiple Tensorflow versions installed. However, the Noori Repository required `tensorflow-gpu-1.15.0`, which was not available on the cluster. Rather than install it on the HPC and worry about configuration, we discovered the existing docker container `tensorflow/tensorflow:1.15.5-gpu`. From here, it was a simple matter of installing the remaining dependencies – since the container existed in isolation, dependency conflicts weren't a concern. This container was used as the runtime environment when creating and evaluating the model.[6]

5.2.2 HDF5

We also had to use the HDF5 module. Compared to singularity, HDF5 was an instance where the advantages of the Lmod system made themselves clear. While the process of installing the latest version of Singularity is not as simple as running `yum` or `apt-get`, it is nevertheless a matter of copying code into the terminal to build from source – since all containers must conform to the OCI standard, Singularity is both backwards and forwards compatible and there is very little, if any, configuration required when installing Singularity.

HDF5, on the other hand, has many important configuration options that, if misconfigured, can prevent the program from working the way it should.

HDF5 stands for ”Hierachical Data Format 5”. It is used for big data storage. It functions as a container for complex, structured and unstructured big data. It provides a hierarchical structure with fast I/O times. The HDF5 api provides a pipeline into this compressed file format. It sees use across multiple industries and scientific fields. However, reading the HDF5 file requires the use of special libraries. In Python, that library is h5py, which provides a direct mapping of the HDF5 API into the Python language. [19]

5.3 Python Libraries

There are a set of libraries that are considered standard, almost mandatory, for any data science project, and with which every data scientist must be familiar with in some capacity. The core data science libraries are numpy andscipy. NumPy and SciPy elevate the mathematical and statistical capabilities of Python to the level of Matlab or Mathematica [4]. In the course of this project we had to use a two uncommon Python libraries. NiBabel is a Python library that is able to parse complex neurological images into NumPy arrays. It is a successor to the PyNiFTI library [12]. The other library we used was PyTables, which is somewhat similar to h5py, inasfar as it provides an interface to HDF5. PyTables provides a very specific set of functionalities while abstracting the functionality of the HDF5 API – as a result, it does not offer the full range of features as h5py. However, it offers speed and built-in compatibility with other important libraries, namely pandas. [15]

6 Results and Reflection

6.1 The Program

The program has three components. First, `prepare_data.py` takes a set of MRI image files, formatted in `.nii.gz` format, partitions them into a training and validation set, and stores them in a single file using the HDF5 format(See Section 5.2.2). Next, `train.py` uses the prepared data and builds a U-Net. This step is incredibly long, resulting from the complex nature of the U-Net as well as the complexity of the data. The predictions were comparatively short, with about one prediction completed every few seconds. In the end, with a per-epoch walltime of around 38 minutes, would be impossible to monitor the program while the neural network was being trained. While the model is supposed to run for 100 epochs, or approximately 3800 minutes, we capped it at 10.

6.1.1 Problems

When setting up the model, we initially ran into problems with the generated HDF5 file. At the end of an epoch, the model would throw a segmentation fault. We narrowed down the cause to illegal parallel access of the data. The piece of the program responsible for validation

[4]You can read more about the Scipy stack at `https://scipy.org/`. To see the full list of libraries used, check out the repository

was trying to access the hdf5 file before the piece responsible for training had released the lock on the file, and the file had been instructed not to allow that type of parallel access. We fixed this by forcing all operations onto the main thread, setting `workers = 0`.

We then tried to increase the training speed by modifying the way the model was constructed. First, we attempted to use `tf.distribute.MirroredStrategy`, which performs synchronous training across multiple GPUs by keeping the variables synchronized. This failed, because the function used for training the model, `fit_generator` was incompatible with distributed strategies. Since we were using a custom data generator, we couldn't replace `fit_generator` with the more versatile `fit`. We next tried to change the model to use `tf.keras.utils.multi_gpu_model`. This ran —but resulted in slowdown, not speedup. This was a result of the way in which the multi_gpu_model works (See Figure 12), and the presence of the HDF5 file lock.

6.2 Results

The model performs incredibly well after just a few epochs (see Figure **??**. The training and validation accuracies are high, and the training loss is low. However, the validation loss is considerably higher than the training loss, and is stubborn. While we could be satisfied with the high accuracies, the high validation loss is indicative of the possibility of overfitting. We shouldn't "cut corners" on a model with real world implications, such as this one. Therefore, it is likely necessary to run the model for the full 100 epochs.

6.3 Concluding thoughts

This model benefits in many ways from the features offered by the HPC cluster. The modularity provided by lmod allows users to take advantage of advanced technologies, such as HDF5, previously used exclusively by engineers, all without the hassle of building from the source repository. Singularity fills a similar, but distinct, role. Instead of environment management, Singularity's isolated container environment eases the process of dependency management, allowing them to build a container exactly to the specifications of their model and ensure that anyone, anywhere will be able to run it. Finally, without a tool like slurm, it would be impossible to run a model such as this: the amount of time to complete is too long to be continuously monitored. At 10 epochs it took over 6 hours. For the full 100, it would take over two days. A personal computer wouldn't be able to allocate the necessary resources to this task; on the other hand, the HPC environment has a number of GPU nodes, which slurm is able to find and assign to the training of the model. We've determined that the use of high performance computing is invaluable in order to solve these kinds of real world problems dealing with sophisticated models and complex data.

Appendix A : Figures

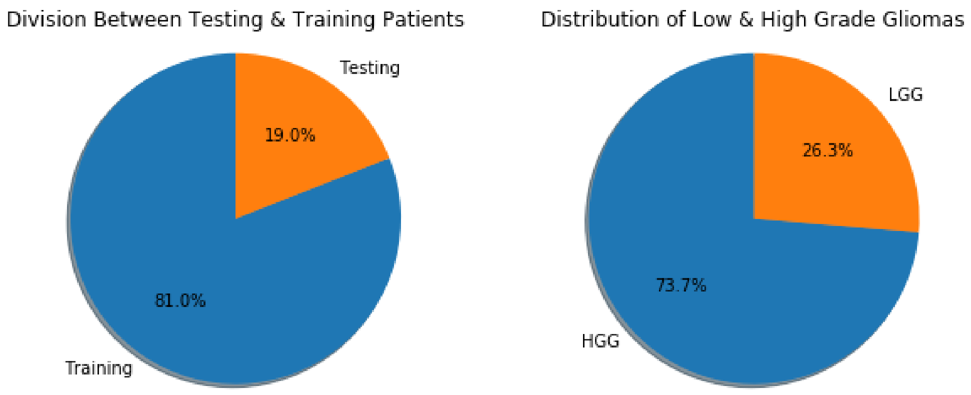


Figure 1: Exploratory data analysis on the BraTS dataset

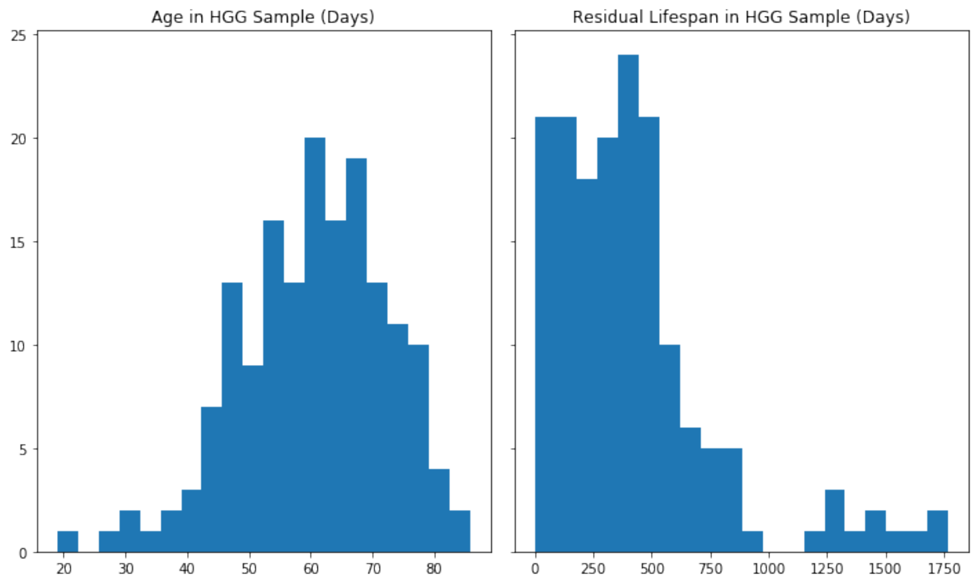


Figure 2: Exploratory data analysis on the BraTS dataset's metadata

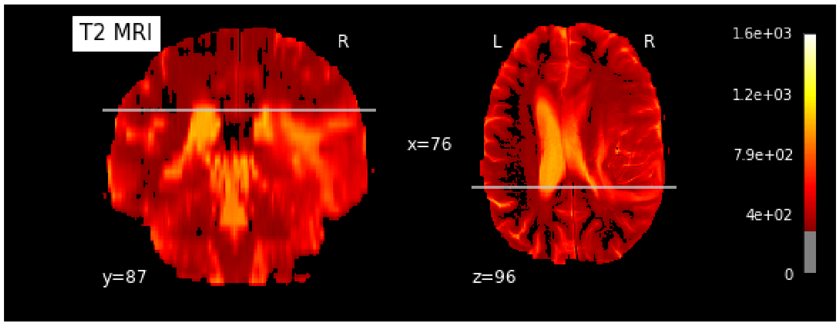
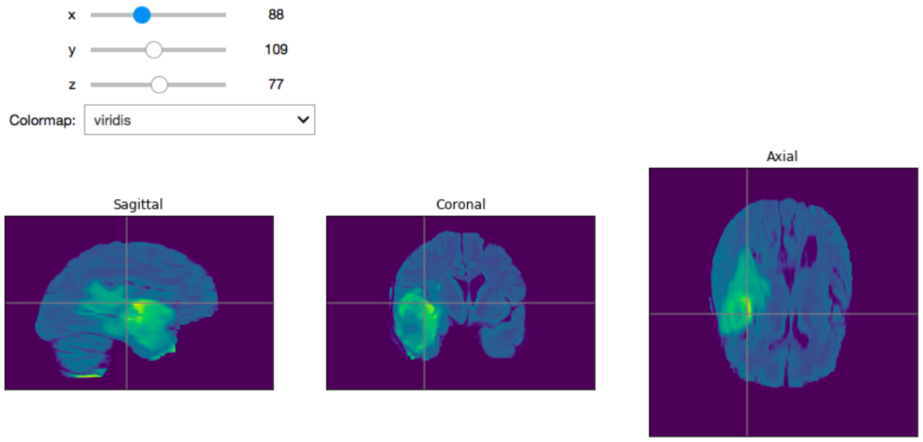


Figure 3: Static images of an MRI in the BraTS dataset



x 88
y 109
z 77
Colormap: viridis

Figure 4: An interactive widget we created for navigating BraTS MRIs in three dimensions

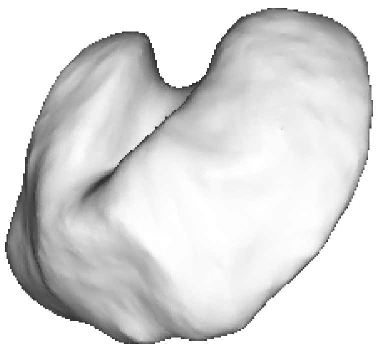


Figure 5: An interactive widget we created for studying the surface of tumors in the BraTS dataset

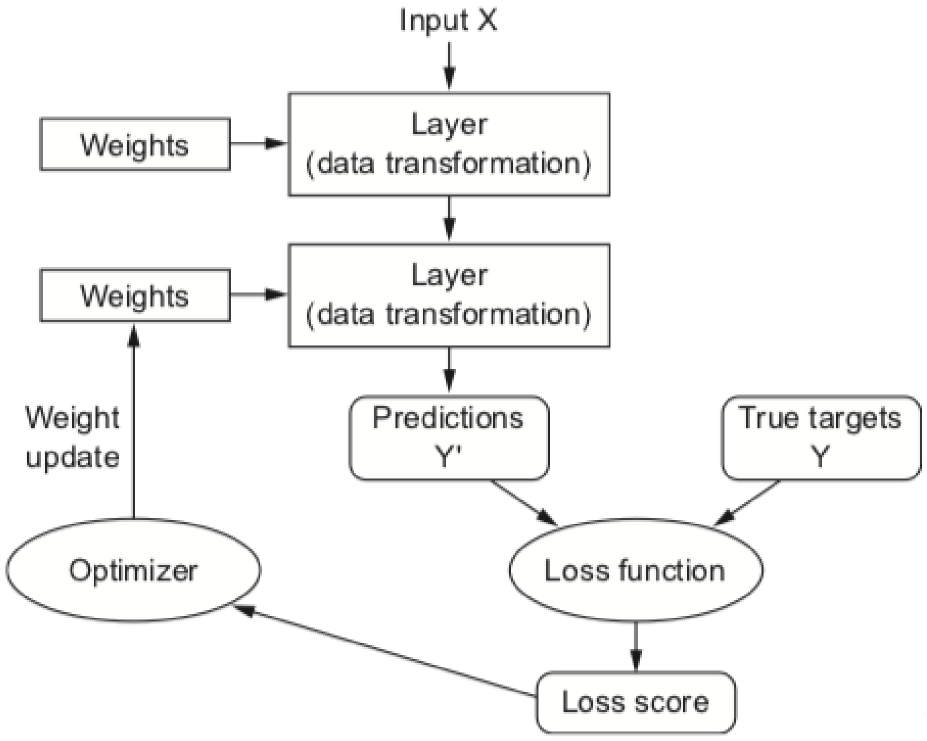


Figure 6: Illustration of the mechanics of a traditional neural network [4]

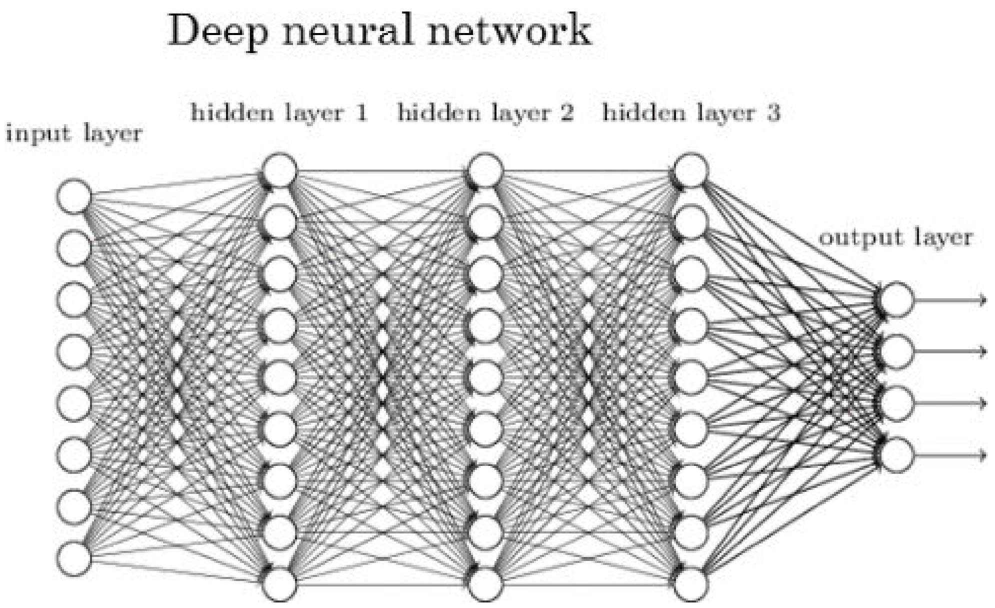


Figure 7: Illustration of the architecture of a deep neural network [4]

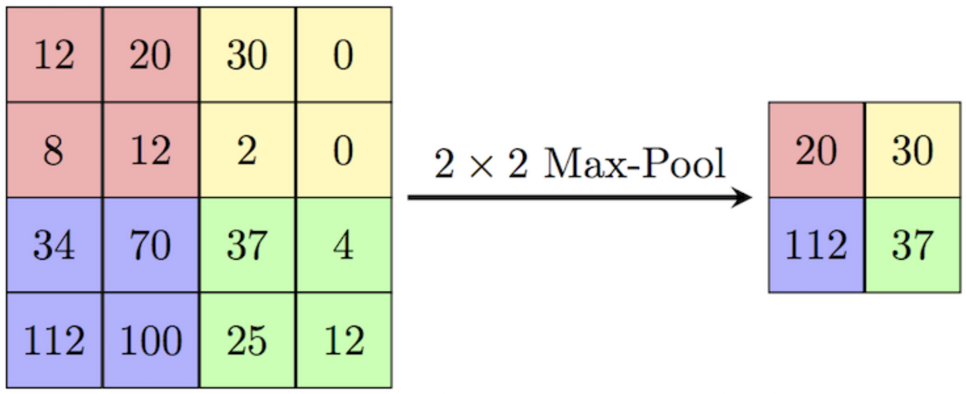


Figure 8: An example of a max-pooling operation

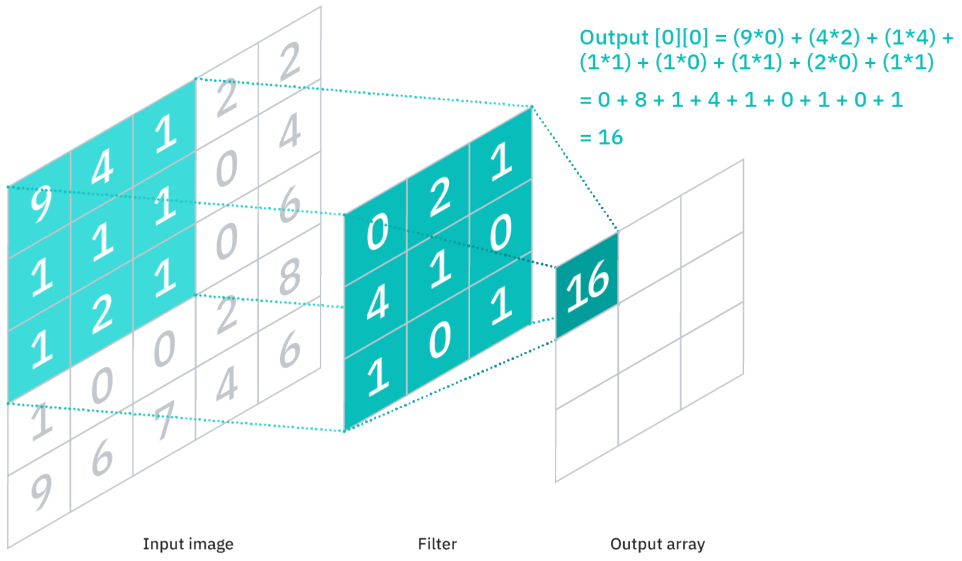

Output [0][0] = (9*0) + (4*2) + (1*4) + (1*1) + (1*0) + (1*1) + (2*0) + (1*1)
= 0 + 8 + 1 + 4 + 1 + 0 + 1 + 0 + 1
= 16
Input image
Filter
Output array

Figure 9: An example of a convolution operation

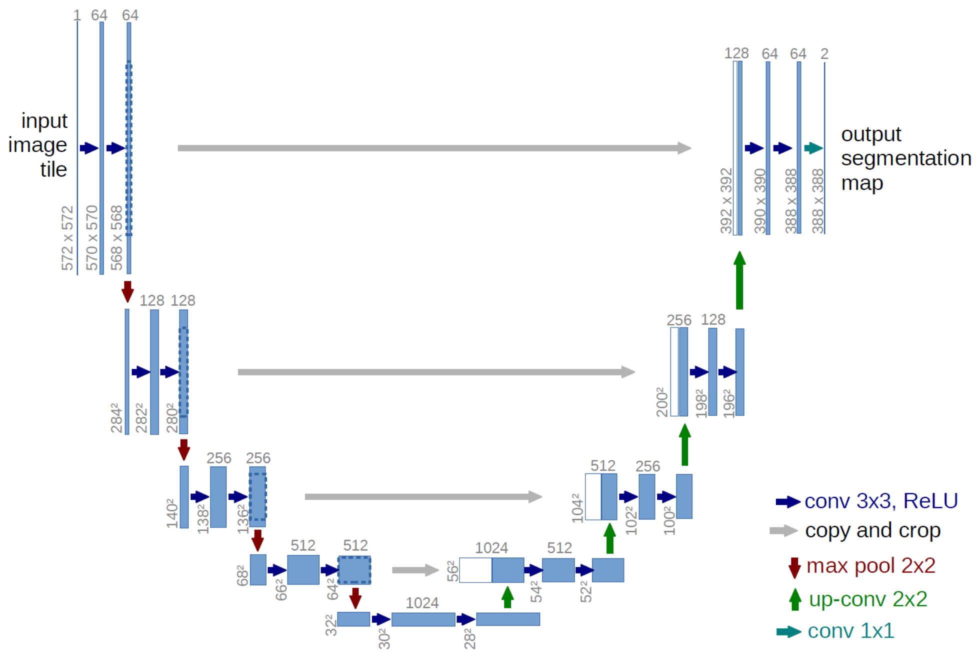


Figure 10: Illustration of the architecture of a U-Net [16]

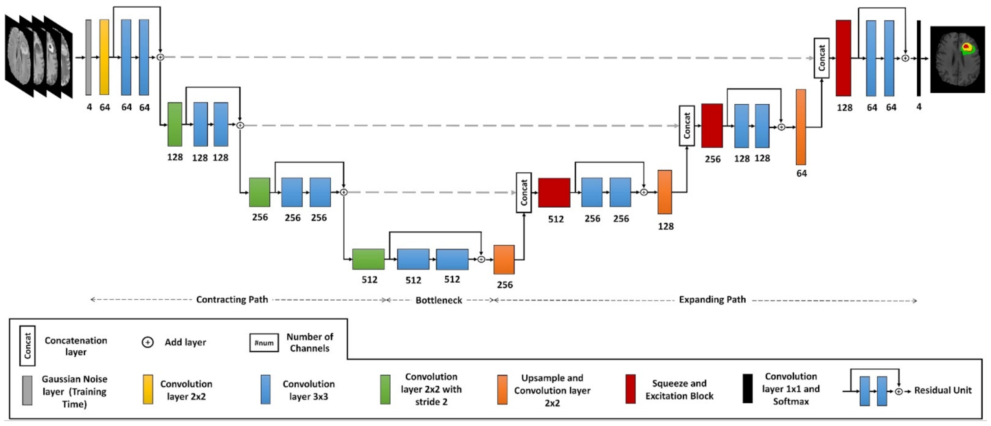


Figure 11: Illustration of the model used in this project [14]

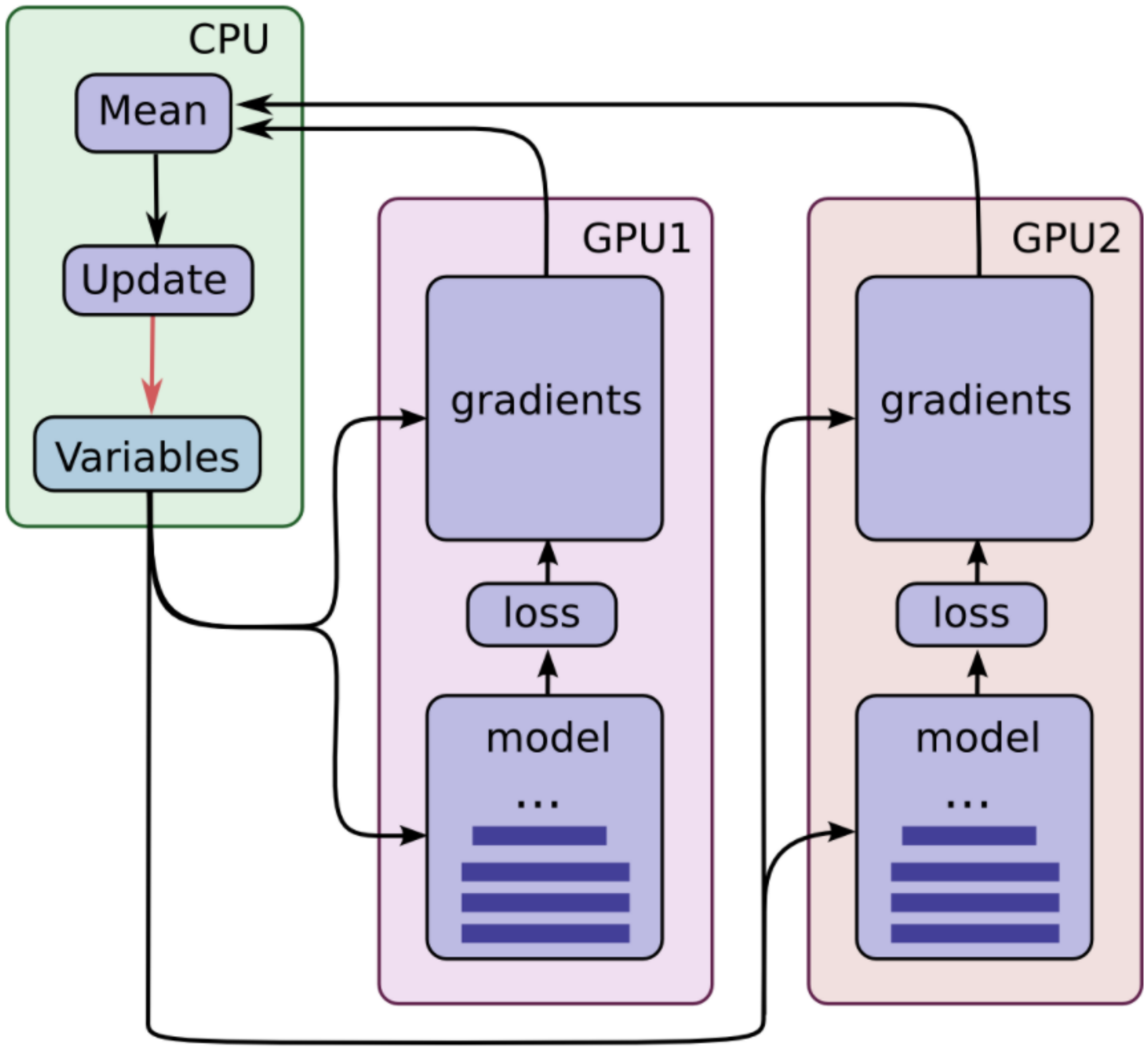
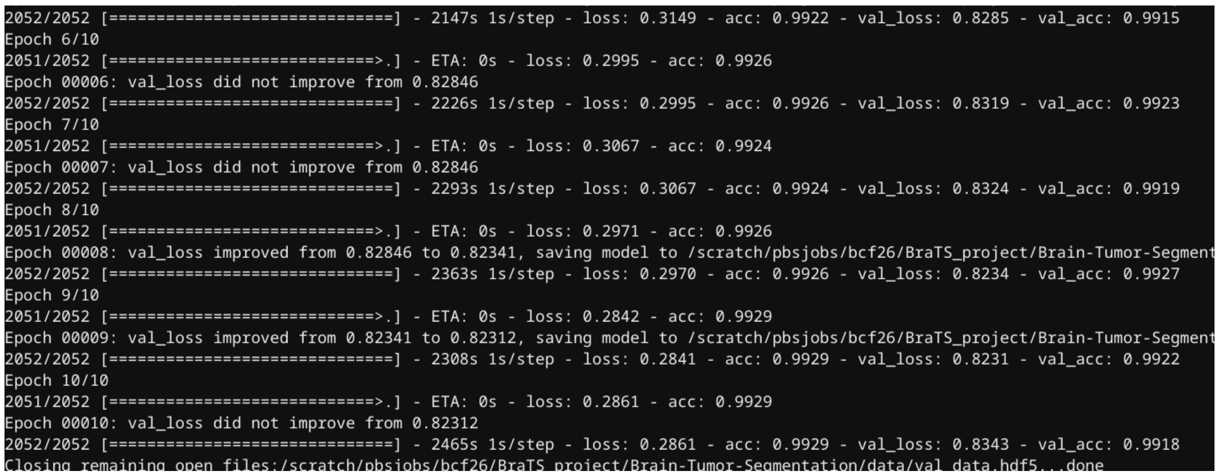


Figure 12: The process in which Tensorflow trains with multiple gpus



```
2052/2052 [==============================] - 2147s 1s/step - loss: 0.3149 - acc: 0.9922 - val_loss: 0.8285 - val_acc: 0.9915
Epoch 6/10
2051/2052 [============================>.] - ETA: 0s - loss: 0.2995 - acc: 0.9926
Epoch 00006: val_loss did not improve from 0.82846
2052/2052 [==============================] - 2226s 1s/step - loss: 0.2995 - acc: 0.9926 - val_loss: 0.8319 - val_acc: 0.9923
Epoch 7/10
2051/2052 [============================>.] - ETA: 0s - loss: 0.3067 - acc: 0.9924
Epoch 00007: val_loss did not improve from 0.82846
2052/2052 [==============================] - 2293s 1s/step - loss: 0.3067 - acc: 0.9924 - val_loss: 0.8324 - val_acc: 0.9919
Epoch 8/10
2051/2052 [============================>.] - ETA: 0s - loss: 0.2971 - acc: 0.9926
Epoch 00008: val_loss improved from 0.82846 to 0.82341, saving model to /scratch/pbsjobs/bcf26/BraTS_project/Brain-Tumor-Segment
2052/2052 [==============================] - 2363s 1s/step - loss: 0.2970 - acc: 0.9926 - val_loss: 0.8234 - val_acc: 0.9927
Epoch 9/10
2051/2052 [============================>.] - ETA: 0s - loss: 0.2842 - acc: 0.9929
Epoch 00009: val_loss improved from 0.82341 to 0.82312, saving model to /scratch/pbsjobs/bcf26/BraTS_project/Brain-Tumor-Segment
2052/2052 [==============================] - 2308s 1s/step - loss: 0.2841 - acc: 0.9929 - val_loss: 0.8231 - val_acc: 0.9922
Epoch 10/10
2051/2052 [============================>.] - ETA: 0s - loss: 0.2861 - acc: 0.9929
Epoch 00010: val_loss did not improve from 0.82312
2052/2052 [==============================] - 2465s 1s/step - loss: 0.2861 - acc: 0.9929 - val_loss: 0.8343 - val_acc: 0.9918
Closing remaining open files:/scratch/pbsjobs/bcf26/BraTS_project/Brain-Tumor-Segmentation/data/val_data.hdf5...done
```

Figure 13: The model after 10 epochs

Appendix B: Code[5]

Singularity Definition File

```
Bootstrap: docker
From: tensorflow/tensorflow:1.15.0-gpu

%post
    pip install --user -r requirements.txt
```

[5]For the full code(including a setup script, which will pull the data and build the singularity image, clone the repository at `https://github.com/bcflock/courseproject_312`

Slurm Script

```
#!/bin/bash
#SBATCH --output /scratch/pbsjobs/bcf26/test.o%j
#SBATCH --nodes 1 #
#SBATCH        --gpus-per-task 1
#SBATCH --time=0-18:00:00
#SBATCH --job-name="312cp"
#SBATCH --mail-user="bcf26@case.edu"
#SBATCH --mem-per-gpu=12G
module load cuda/8.0 singularity/3.5.1 hdf5/1.10.1 python

mkdir $PFSDIR/course-project
WORKDIR= $PFSDIR/course-project
mkdir $WORKDIR/Brain-Tumor-Segmentation
mkdir $WORKDIR/Brain-Tumor-Segmentation/data

cp -R ./data $WORKDIR/Brain-Tumor-Segmentation
find . -type f ! -name "*.py*" -exec cp {} $WORKDIR/
cp *.py $WORKDIR/Brain-Tumor-Segmentation

singularity exec --nv $WORKDIR/tf.sif
    python $WORKDIR/Brain-Tumor-Segmentation/prepare_data.py
singularity exec --nv $WORKDIR/tf.sif
    python $WORKDIR/Brain-Tumor-Segmentation/train.py
singularity exec --nv $WORKDIR/tf.sif
    python $WORKDIR/Brain-Tumor-Segmentation/predict.py
```

Modified Noori Code

In the course of this project, two files from the Noori repository were modified. The rest of the repository, as it has been unmodified, can be found at `https://github.com/Mehrdad-Noori/Brain-Tumor-Segmentation`

train.py

```
import os
import threading
import tables
import faulthandler; faulthandler.enable();
import numpy as np
#from mpi4py import MPI
from config import cfg
from model import unet_model
from data_generator import CustomDataGenerator
from tensorflow.keras.callbacks import CSVLogger, ModelCheckpoint, TensorBoard
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import utils
from tensorflow.keras.utils import multi_gpu_model
def train_model(hdf5_dir, brains_idx_dir, view, modified_unet=True,
                    batch_size=16, val_batch_size=32,
                    lr=0.01, epochs=100, hor_flip=False,
                    ver_flip=False, zoom_range=0.0, save_dir='./save/',
                    start_chs=64, levels=3, multiprocessing=False,
                    load_model_dir=None):
    """
    The function that builds/loads UNet model,
    initializes the data generators for training
    and validation, and finally trains the model.
    """
    # preparing generators
    hdf5_file        = tables.open_file(hdf5_dir, mode='r+')
    brain_idx        = np.load(brains_idx_dir)
    datagen_train    = CustomDataGenerator(
                                    hdf5_file,
                                    brain_idx,
                                    batch_size,
                                    view,
                                    'train',
                                    hor_flip,
                                    ver_flip,
                                    zoom_range,
                                    shuffle=True
                                    )
    datagen_val      = CustomDataGenerator(
                                hdf5_file,
                                brain_idx,
                                val_batch_size,
                                view,
                                'validation',
                                shuffle=False
                                )
    # add callbacks
    save_dir     = os.path.join(save_dir, '{}_{}'.format(
                        view,
                        os.path.basename(brains_idx_dir)[:5])
                    )
    if not os.path.isdir(save_dir):
        os.mkdir(save_dir)
    logger       = CSVLogger(os.path.join(save_dir, 'log.txt'))
    checkpointer = ModelCheckpoint(
                            filepath = os.path.join(save_dir, 'model.hdf5'),
                            verbose=1, save_best_only=True)
    tensorboard  = TensorBoard(os.path.join(save_dir, 'tensorboard'))
    callbacks    = [logger, checkpointer, tensorboard]

    #with strategy.scope():
    # building the model
    model_input_shape = datagen_train.data_shape[1:]
    model             = unet_model(model_input_shape, modified_unet,
                                lr, start_chs, levels)
    # training the model
    model.fit_generator(datagen_train,
                        epochs=epochs,
                        use_multiprocessing=multiprocessing,
                        callbacks=callbacks,
                        validation_data = datagen_val,
                        workers = 0
        )



if __name__ == '__main__':
    train_model(
                cfg['hdf5_dir'],
                cfg['brains_idx_dir'],
                cfg['view'],
                cfg['modified_unet'],
                cfg['batch_size'],
                cfg['val_batch_size'],
                cfg['lr'],
                cfg['epochs'],
                cfg['hor_flip'],
                cfg['ver_flip'],
                cfg['zoom_range'],
                cfg['save_dir'],
                cfg['start_chs'],
                cfg['levels'],
                cfg['multiprocessing'],
                cfg['load_model_dir']
                )
```

config.py

```
import os

"""
The required configurations for training phase ('prepare_Data.py', 'train.py').
"""

cfg = dict()

"""
The coordinates to crop brain volumes. For example, a brain volume with the
One can set the x0,x1,... by calculating none zero pixels through dataset.
Note that the final three shapes must be divisible by the network downscale rate.
"""
cfg['crop_coord']            =  {'x0':42, 'x1':194,
                                 'y0':29, 'y1':221,
                                 'z0':2,  'z1':146}

"""
The path to all brain volumes (ex: suppose we have a folder
'MICCAI_BraTS_2019_Data_Training'
that contains two HGG and LGG folders so:
data_dir='./MICCAI_BraTS_2019_Data_Training/*/*')
"""
cfg['data_dir']              = '''
    ./Brain-Tumor-Segmentation/data/MICCAI_BraTS_2018_Data_Training/*/*
    '''

"""
The final data shapes of saved table file.
"""
cfg['table_data_shape']      =  (cfg["crop_coord"]['z1']-cfg["crop_coord"]['z0'],
                                 cfg["crop_coord"]['y1']-cfg["crop_coord"]['y0'],
                                 cfg["crop_coord"]['x1']-cfg["crop_coord"]['x0'])

"""
BraTS datasets contain 4 channels: (FLAIR, T1, T1ce, T2)
```

```
"""
cfg['data_channels']         = 4

"""
The path to save table file + k-fold files
"""
cfg['save_data_dir']         = './Brain-Tumor-Segmentation/data/'

"""
The path to save models + log files + tensorboards
"""
cfg['save_dir']                        = './Brain-Tumor-Segmentation/save/'

"""
k-fold cross-validation
"""
cfg['k_fold']                = 5

"""
The defualt path of saved table.
"""
cfg['hdf5_dir']              = './Brain-Tumor-Segmentation/data/val_data.hdf5'

"""
The path to brain indexes of specific fold
(a numpy file that was saved in ./data/ by default)
"""
cfg['brains_idx_dir']        = './Brain-Tumor-Segmentation/data/fold0_idx.npy'

"""
'axial', 'sagittal' or 'coronal'. The 'view' has no effect in "prepare_data.py".
All 2D slices and the model will be prepared  with respect to 'view'.
"""
cfg['view']                  = 'axial'

"""
The batch size for training and validating the model
"""
cfg['batch_size']            = 16#64
cfg['val_batch_size']        = 32

"""
The augmentation parameters.
"""
cfg['hor_flip']              = True
cfg['ver_flip']              = True
cfg['rotation_range']        = 0
```

```
cfg['zoom_range']            = 0.

"""
The leraning rate and the number of epochs for training the model
"""
cfg['epochs']                = 10#100
cfg['lr']                    = 0.008

"""
If True, use process-based threading. "https://keras.io/models/model/"
"""
cfg['multiprocessing']       = True

"""
Maximum number of processes to spin up when using process-based threading.
If unspecified, workers will default to 1. If 0, will execute the generator
on the main thread. "https://keras.io/models/model/"
"""
cfg['workers']               = 0

"""
Whether to use the proposed modifid UNet or the original UNet
"""
cfg['modified_unet']         = True

"""
The depth of the U-structure
"""
cfg['levels']                = 3

"""
The number of channels of the first conv
"""
cfg['start_chs']             = 64

"""
If specified, before training, the model weights
will be loaded from this path otherwise
the model will be trained from scratch.
"""
cfg['load_model_dir']        = None
```

References

[1] Bruce Alberts. *Molecular biology of the cell.* 2008.

[2] Stefan Bauer et al. “A survey of MRI-based medical image analysis for brain tumor studies”. In: *Physics in Medicine & Biology* 58.13 (2013), R97.

[3] Yoshua Bengio. “Learning Deep Architectures for AI”. In: *Found. Trends Mach. Learn.* 2.1 (Jan. 2009), pp. 1–127. ISSN: 1935-8237. DOI: `10.1561/2200000006`. URL: `https://doi.org/10.1561/2200000006`.

[4] François Chollet. *Deep Learning with Python.* 1st ed. Manning Publications, 2017.

[5] Michael Crosby. *Containerd: a daemon to control runC.* URL: `https://www.docker.com/blog/containerd-daemon-to-control-runc/`. [Accessed: 2021-05-11].

[6] Sanjaya Gajurel et al. *Case Western Reserve HPC.* URL: `https://sites.google.com/a/case.edu/hpcc/home`. [Accessed: 2021-05-13].

[7] R. George et al. *MRI Protocols, MRI Planning, MRI Techniques and Anatomy.* [Accessed 2021-05-13]. URL: `%5Curl%7Bhttps://mrimaster.com/physics%5C%20intro.html%7D`.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105.

[9] Y. Lecun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: `10.1109/5.726791`.

[10] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), pp. 436–444.

[11] Bjoern H. Menze et al. “The Multimodal Brain Tumor Image Segmentation Benchmark (BRATS)”. In: *IEEE Transactions on Medical Imaging* 34.10 (2015), pp. 1993–2024. DOI: `10.1109/TMI.2014.2377694`.

[12] NiBabel Development Team. *nipy/nibabel: 3.0.1.* Version 3.0.1. Jan. 2020. DOI: `10.5281/zenodo.3628482`. URL: `https://doi.org/10.5281/zenodo.3628482`.

[13] Nils J. Nilsson. *Introduction to Machine Learning.* Nov. 3, 1998.

[14] Mehrdad Noori, Ali Bahri, and Karim Mohammadi. “Attention-Guided Version of 2D UNet for Automatic Brain Tumor Segmentation”. In: *2019 9th International Conference on Computer and Knowledge Engineering (ICCKE).* IEEE. 2019, pp. 269–275.

[15] PyTables Developers Team. *PyTables: Hierarchical Datasets in Python.* 2002. URL: `http://www.pytables.org/`.

[16] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation.* 2015. arXiv: `1505.04597 [cs.CV]`.

[17] SchedMD. *Slurm Documentation.* URL: `https://slurm.schedmd.com/`. [Accessed: 2021-05-13].

[18] Sylabs Inc. *User Guide – Singularity Container.* URL: `https://sylabs.io/guides/3.0/user-guide/index.html`. [Accessed: 2021-05-1].

[19] The HDF5 Group. *The HDF5 ® Library and File Format*. URL: `https://www.hdfgroup.org/solutions/hdf5/`. [Accessed: 2021-05-12].