Rose-Hulman Institute of Technology

Rose-Hulman Scholar

Graduate Theses - Electrical and Computer Engineering

Electrical and Computer Engineering

Summer 7-2020

# Inter-slice Compression and Reconstruction of Glioma Magnetic Resonance Imaging (MRI) Data Using Encoder-Decoder Neural Networks

Gavin Michael Karr

Follow this and additional works at: https://scholar.rose-hulman.edu/dept_electrical

Part of the Electrical and Electronics Commons

**Inter-slice Compression and Reconstruction of Glioma Magnetic Resonance Imaging (MRI) Data Using Encoder-Decoder Neural Networks**

A Thesis

Submitted to the Faculty

Of

Rose-Hulman Institute of Technology

by

Gavin Michael Karr

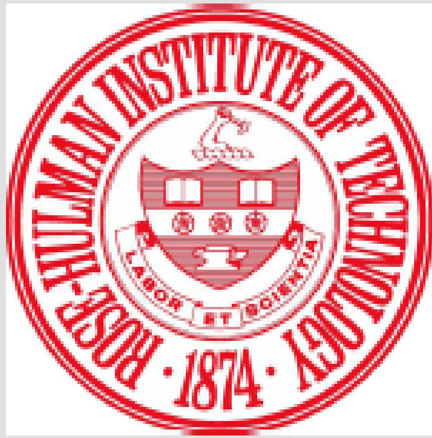
In Partial Fulfillment of the Requirements for the Degree

of

Master of Science in Electrical Engineering

July 2020

© 2020 Gavin Michael Karr




# ROSE-HULMAN INSTITUTE OF TECHNOLOGY

## Final Examination Report

Gavin Karr
Name

Electrical Engineering
Graduate Major

Thesis Title Inter-slice Compression and Reconstruction of Glioma MRI Data Using Encoder-Decoder Neural Networks

**DATE OF EXAM:** July 7, 2020

**EXAMINATION COMMITTEE:**

| | Thesis Advisory Committee | Department |
|---|---|---|
| Thesis Advisor: | Michael McInerney | PHOE |
| | Kurt Bryan | MA |
| | Matthew Boutell | CSSE |
| | | |
| | | |

**PASSED** X **FAILED** ____

## ABSTRACT

Karr, Gavin Michael

M.S.E.E

Rose-Hulman Institute of Technology

July 2020

Inter-slice Compression and Reconstruction of Glioma Magnetic Resonance Imaging (MRI) Data Using Encoder-Decoder Neural Networks

Thesis Advisor: Dr. Michael McInerney

Magnetic Resonance Imaging (MRI) scans of patients with brain tumors are an important source of pre-surgical medical information. These three-dimensional image volumes can be represented as a stack of two-dimensional image slices.

The objective of this thesis is to compress the size of these image volumes by removing the odd-numbered slices and reconstruct the image volume using an encoder-decoder convolutional neural network. This neural network architecture is based on a modified form of the U-net segmentation network, which has been adjusted to allow for multiple image inputs and to support a network capable of generating new image slices. A novel method of performing slice interpolation is introduced in which the image features extracted by the neural network are averaged at each network layer to form the intermediary slice from the two input slices.

The MRI volume reconstruction performed by the encoder-decoder neural network is compared against linear interpolation of the image slices, and the metric used is the peak signal-to-noise-ratio. The reconstruction of the volume by the neural network slightly underperforms the linear interpolation baseline due to both methods being close to optimal in performance. Overall, the reconstruction quality of both methods is high since the initial slice distance causes

little variation between adjacent slices. This thesis concludes that the neural network method of compression and reconstruction has potential in cases where inter-slice resolution is initially poor, such as at 4 millimeters and higher, while linear interpolation is sufficient at resolutions below 4 millimeters.

Keywords: Deep Learning, Compression, Medical Imaging, Convolutional Neural Networks

# TABLE OF CONTENTS

## Contents

LIST OF FIGURES ..... iv
LIST OF TABLES ..... v
LIST OF ABBREVIATIONS ..... vi
1. INTRODUCTION ..... 1
2. BACKGROUND ..... 2
2.1 Overview of Gliomas ..... 2
2.2 Overview of Medical Imaging ..... 2
2.3 Overview of Classification Convolutional Neural Networks (CNN) ..... 5
2.4 Relation to Image Segmentation ..... 7
2.5 Detailed Neural Network Techniques ..... 11
2.6 Reason for Selecting the Encoder-Decoder Structure for the Neural Network ..... 15
3. PROPOSAL FOR INVESTIGATING SLICE INTERPOLATION ..... 17
3.1 Data Set Selection and Preprocessing ..... 18
4. INTERPOLATION NEURAL NETWORK ARCHITECTURE ..... 20
4.1 Interpolation Network Experimental Configurations ..... 23
5. INTERPOLATION RESULTS ..... 26
5.1 Segmentation Results of the Interpolated MRI Volumes ..... 31
6. POTENTIAL ISSUES WITH PROPOSED INTERPOLATION APPROACH ..... 33
7. CONCLUSION ..... 35
8. FUTURE IMPROVEMENTS ..... 37
LIST OF REFERENCES ..... 39
Appendix A: Neural Network Model A Code ..... 43
Appendix B: Neural Network Model C Code ..... 46
Appendix C: Neural Network Training Code (Jupyter Notebook Format) ..... 49
Appendix D: Test Neural Network Performance (Jupyter Notebook Format) ..... 53

# LIST OF FIGURES

Figure 1: Medical Image Perspectives ........ 3
Figure 2: Segmentation Labels for the BraTS dataset ........ 4
Figure 3: Classic Classification CNN ........ 5
Figure 4: U-net Neural Network Model ........ 9
Figure 5: LeakyReLU Activation Function ........ 14
Figure 6: ReLU Activation Function ........ 14
Figure 7: Example layout of interpolation model architecture with channel averaging residual skip connections ........ 21
Figure 8: Histogram of BraTS Dataset Maximum Pixel Intensities ........ 22
Figure 9: Example MRI Volume Slice Interpolation Result. ........ 28
Figure 10: Slice Reconstruction at 1-Slice Distance ........ 30
Figure 11: Slice Reconstruction at 5-Slice Distance ........ 30

# LIST OF TABLES

Table 1: Slice Reconstruction PSNR of Tested Interpolation Methods ....................................... 27
Table 2: PSNR of Interpolation Methods at Varying Input Slice Distances ................................ 29
Table 3: Dice Coefficient for Tumor Segmentation of Processed Datasets ................................. 31

# LIST OF ABBREVIATIONS

| | |
|---|---|
| BraTS | Brain Tumor Segmentation Challenge |
| CNN | Convolutional Neural Network |
| FLAIR | Fluid Attenuated Inversion Recovery |
| MRI | Magnetic Resonance Imaging |
| PSNR | Peak Signal-to-Noise Ratio |
| ReLU | Rectified Linear Unit |

# 1. INTRODUCTION

The field of medical imaging is dependent on affordable access to high-quality imaging, often with limited time and financial resources. Magnetic Resonance Imaging (MRI) technology highlights these qualities due to the relatively long scan times, high cost of the imaging equipment, and the trained professionals necessary to interpret and make decisions. Due to the difficulty of preoperative diagnosis and the risk associated with brain surgery, pre-treatment patients with gliomas were selected as the subject for this thesis.

Compressing and then later reconstructing the images from an MRI scan in high quality has the potential to lower the resource utilization of acquiring and maintaining databases of patient scans. This thesis explores the use of deep learning algorithms to perform the reconstruction of compressed MRI scans. The performance of the implemented encoder-decoder deep learning model is compared against linear averaging and evaluated by the amount of information lost in the reconstruction process. Additionally, an established deep learning algorithm that performs segmentation analysis of MRI scans is reproduced and used to study how the compression and reconstruction process impacts the performance of segmentation models that derive their performance metrics from image content rather than just image quality.

In order to provide a foundation for discussing the impact of modern advancements in the rapidly evolving field of deep learning, this thesis also discusses the impact on performance and efficiency of modern methods in the deep learning modeling and training process.

# 2. BACKGROUND

## 2.1 Overview of Gliomas

Gliomas are a type of fast-growing brain tumor which arise from the glial cells in the brain. As one of the most common cells in the central nervous system, glial cells support and protect neurons throughout the brain and spinal cord. Due to the abundance of these cells, gliomas are defined by their diffuse infiltration of the brain tissue and high vascularization [1]. Therefore, they are usually difficult to access surgically when compared to other cancers. In the late stage, gliomas tend to form tendrils of malignant tissue that can span the cranium. Gliomas are graded on a scale of one to four, which describes the presence of faster growing and malignant cells within the tumor [2]. Grade I and II tumors are called low-grade gliomas which show lower growth rate and well-defined tumor boundaries. Grade III and IV tumors are further categorized as high-grade gliomas, which show the presence of highly malignant tissue and often have poorly defined tumor boundaries, creating and spreading along blood vessels in the brain and spinal cord. Additionally, high-grade gliomas nearly always reoccur, even after surgical resection and accompanying standard of care treatments. Overall survival time for these patients is typically between 12 and 18 months, with less than one in twenty patients surviving longer than five years [1]. Due to the incurable nature of the disease and the fast progression of the disease, patients require regular and frequent monitoring to identify reoccurrences and plan interventions [3].

## 2.2 Overview of Medical Imaging

A non-invasive imaging technique used for pre-surgical diagnosis and post-treatment monitoring is magnetic resonance imaging. MRI technology allows healthcare professionals to

form a volume model of the brain from a series of image slices. A typical scan of the brain lasts 45 minutes, after which the results are sent to a radiologist for annotation, often taking several days [4]. MRI scans for a 1.5 to 3 Tesla scanner have typical resolutions of 1.5 mm by 1.5 mm in the image plane, and 4 mm in the inter-slice direction [5]. Note that the resolution between each slice is poorer than the resolution between pixels in the image slice itself.

The image plane in which a 2D slice resides depends on how the sampled information is viewed. The three standard 2D views are the axial, coronal, and sagittal views.

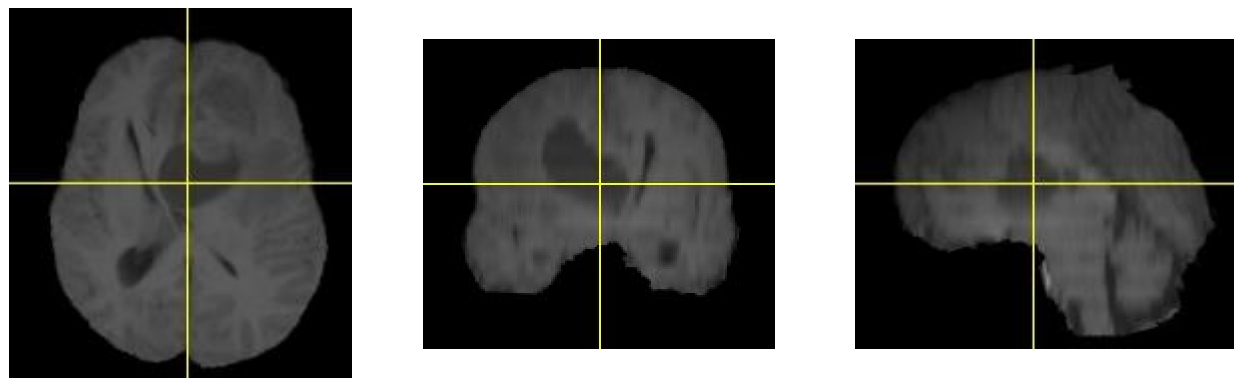


**Figure 1: The above three images are slices of a brain MRI volume from the BraTS2018 dataset, take from a different 2D view. The yellow axes indicate the point in origin in 3D space. From left to right, the views are axial, coronal, and sagittal.**

Medical MRI data is usually stored as a volume from which 2D views are sampled. However, differences in the anatomies of individual patients as well as how the scans are acquired can complicate the reconstruction of the volume from the slices, as well as comparison between patients. As raw data, the scans cannot be compared point-by-point and there is no guarantee that the scans of two patients are geometrically aligned [6].

A process to overcome this variance and create a coordinate system supporting scans taken at different times and from different patients is known as image registration. Using MRI

scans of brains as an example, registration allows for a common coordinate system to be drawn for an individual scan so that the scan of one patient's brain can be geometrically and anatomically compared to the brain scans of other patients who may have been tested at another time or location. This professionally standardized knowledge is referred to as an anatomical atlas, which produces a standardized coordinate system allowing for comparison of anatomical structures between institutions [6].

A key part of the tumor grading and diagnosis process is determining the tissue makeup of the tumor. The final tumor grading is done post-surgery, however, it is important to identify the size, shape, and tissue makeup of gliomas prior to surgery [3]. The tumor shape and location in the image volumes are used to identify possible complications and map out boundaries for computer-assisted surgical devices. Segmenting the image into categories such as whole tumor, tumor core, and enhancing tumor core further aids medical professionals in this decision-making process.

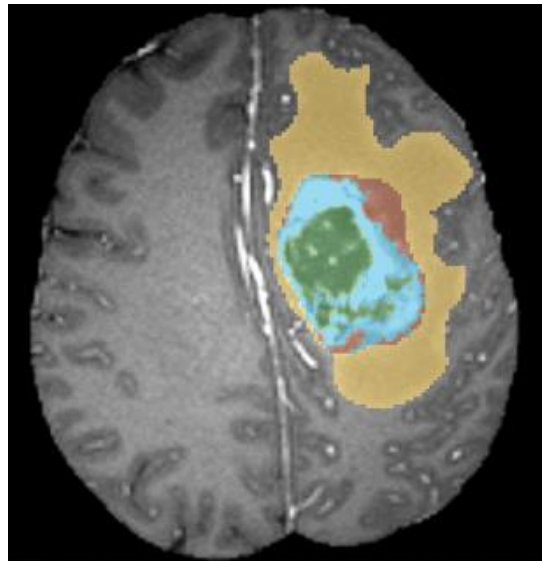


**Figure 2: Segmented labels for peritumoral edema (ED - yellow), non-enhancing tumor core (ET – red), and enhancing tumor (NCR/NET – Blue/Green)**

The manual segmentation of the MR image is aided by the different imaging modalities used in the data collection process. An example collection of modalities would be an imaging sequence taken and used by the BraTS data challenge gathered by the University of Pennsylvania School of Medicine: pre-contrast T1 (T1), post-contrast T1-weighted (T1Gd), T2-weighted (T2), and Fluid Attenuated Inversion Recovery (FLAIR) [7]. Together, the multimodal scans can be used to highlight different types of tissue features [8]. Pre-contrast T1 images indicate high-intensity pixels as blood products, mineralization, fat, and melanin. T1Gd images indicate a breakdown of the blood-brain barrier. T2 images highlight non-enhancing tumor, while FLAIR indicates both non-enhancing tumor and peritumoral edema.

## 2.3 Overview of Classification Convolutional Neural Networks (CNN)

The basic principles of CNNs is to combine the ability of convolution to summarize spatially related quantities, such as locations on a graph or pixels in an image, with the feature extraction capabilities of neural networks. A simple supervised CNN used for image classification will consist of a convolutional layer, a pooling layer, and a fully connected layer.

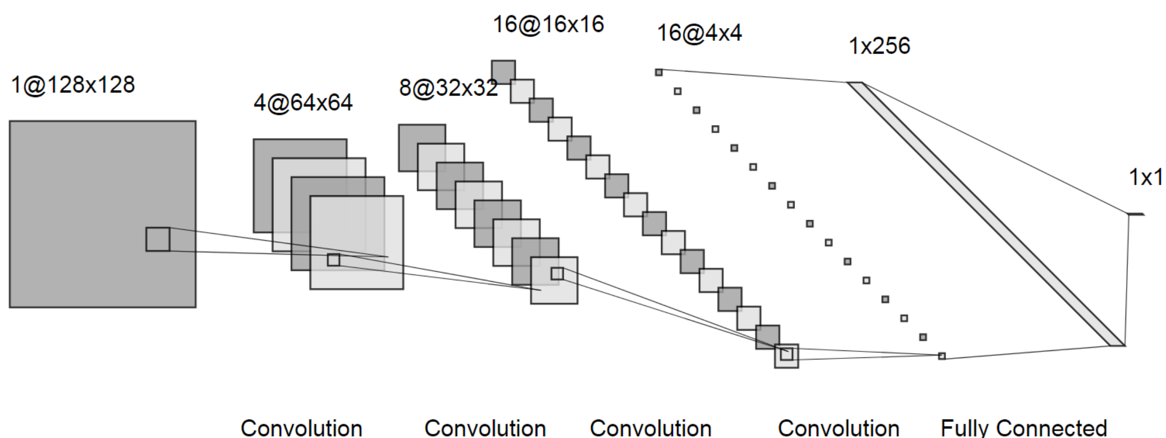



**Figure 3: A diagram showing the reduction of dimensions across a classic classification CNN with labels identifying the filter depth and resolution respectively**

The aforementioned CNN works as follows: An input is first selected and fed into the initial convolutional layer where a small convolutional mask is applied across the image at each pixel. The output of this layer now consists of distinct features that describe the original image based on the weights of the convolutional mask.

This feature map is then fed into a pooling layer, which serves to downsample the image. Additional convolutional and pooling layers can be applied to further reduce the dimension of the feature map. This also serves to increase the receptive field of the network as the original convolutional mask is only able to extract features from a small local region around each pixel. However, following each pooling layer the same sized mask will be able to effectively summarize a larger portion of the original input image through application to the smaller feature map.

This process can be repeated, and eventually, the feature map will be fed into a fully connected neural network in which the neuron of each layer is connected to every neuron in the following layers. This last portion of the network then performs the desired image classification by choosing the classification with the highest activation value in a final activation layer, often using the softmax function.

Back-propagation is commonly used to train such a network, finding effective weights and biases which allow for adequate classification performance. Back-propagation is based on stochastic gradient descent. In this method, initial values for the weights and biases (parameters) are randomly chosen and then a batch of images is fed through the network. The output is collected and compared to the expected output through the use of a loss function such as cross-entropy. This loss is then used to calculate how “wrong” the network was so that the parameters

can be updated accordingly to minimize the loss function. The selection of the training set is stochastic. Using the entire set of training images at once places a large memory burden on the computer. This memory footprint can be lowered by selecting a small random batch of images from the training set and performing the gradient descent process with each of these “mini-batches”. Mini-batches are selected and run through the training process until all images have been used, concluding one epoch of training. Many epochs may be run depending on the learning rate of the training hyperparameters as well as the desire to avoid overfitting the model to the training data. Although each mini-batch does not represent the entire population of data, over the course of a full epoch the average adjustments to the network caused by training on each mini-batch will approximate training on all of the data at once [9].

### 2.4 Relation to Image Segmentation

The manual segmentation process is performed by a radiologist and is used as an expert annotation of a scan to inform the treatment team of the relevant information present in a scan, as well as to point out any anomalies. A significant amount of work has been done in the past to automate this process using machine learning and computer vision techniques with the goal of augmenting the radiologist’s work and increasing the accuracy of the reports. Public competitions such as the BRATS challenge from the University of Pennsylvania seek to not only identify the state of the art segmentation algorithms for brain tumor MRIs, but also to estimate patient survival given the information present in a series of MRI scans. The preeminent work towards this task is increasingly done by convolutional neural networks (CNNs) [10].

The previously described classification network is capable of making a single decision on the topic or contents of a single image, however, it does not describe specific locations or pixels in the image. The separate task of segmenting an image by classifying individual pixels in the

image allows for further analysis of the results. However, the fully connected network at the end of the described simple CNN incorporates dimensionality reduction, and by design, cannot classify each pixel due to the downsampling process as not enough information remains to describe individual pixels.

A network architecture presented in 2015 by the University of Freiburg, called U-net, has tackled this issue and become one of the state of the art segmentation CNN architectures [11]. Developed for use in biomedical imaging, the authors take the basic architecture of a CNN and remove the fully connected network at the output layer. The remaining network is referred to as an “encoder”. This leaves a refined feature map as the remaining layer in the network, known as the “bottleneck layer”. Then, the preceding convolutional and downsampling layers are mirrored in a series of upsampling and deconvolutional layers of the same dimension as the original CNN. This mirrored network is the “decoder”, which is responsible for recovering or generating data given the features from the bottleneck layer. At the output of the decoder is a map matching the dimensions of the input image. The original purpose of this architecture was to generate an entire segmented image using a single end-to-end model, rather than operating a model on single pixels or small groups of pixels.

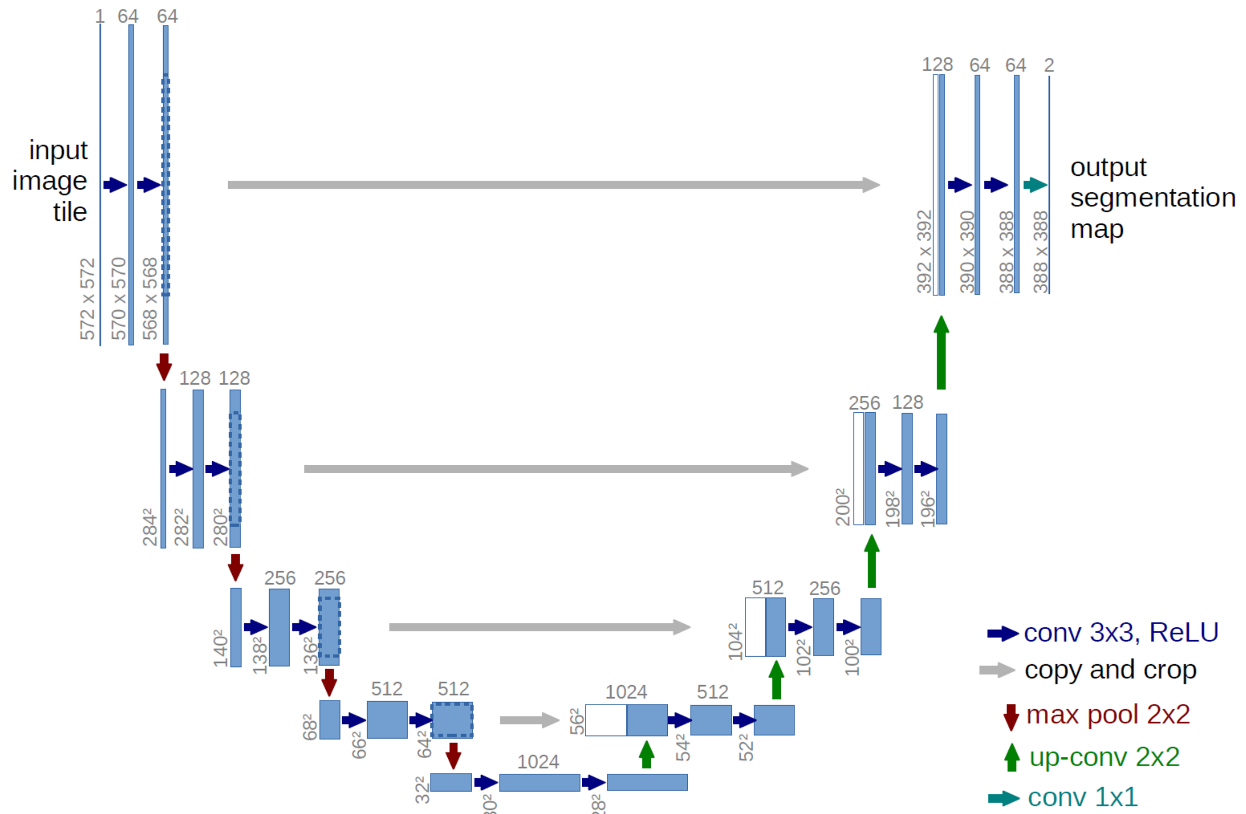



**Figure 4: Diagram of the U-net architecture [11]. Each group of boxes represents a convolutional layer in the network and the corresponding change in resolution. The dashed lines indicate the layer being passed over the skip connection and concatenated. The concatenation is identified by the joining of the white and blue boxes.**

The information lost due to downsampling in the encoder cannot be recovered using upsampling alone in the decoder stage. In order to address this lost information, skip connections are used, shown as grey arrows in Figure 3. These skip connections pull forward the output of a hidden layer from the first convolutional layers and concatenate the extracted information with the corresponding deconvolution, skipping one or more hidden layers. The skip connections then preserve some of the spatial information between layers, allowing both global and local information from the input image to be included in the image reconstruction.

While the training process is the same as the simple CNN, the loss function must be changed to account for the change in the objective of the network from classification to segmentation. The goal of segmentation is to classify individual pixels, so a loss function which compares pixel values directly between the segmented output and the ground truth is desired. Additionally, it can be advantageous to normalize this comparison over the number of pixels in the segmented image. This prevents disproportionately sized object classes from being heavily prioritized by the network. A common loss function in medical image segmentation which accomplishes these goals is based on the Dice coefficient [12]:

$$Dice\ Coefficient = \frac{2|X \cap Y|}{|X| + |Y|}$$

The Dice Coefficient is calculated from two sets of data points or images, the segmented set and the ground truth set. These sets are denoted by $X$ and $Y$ in the equation above. The intersection of the two sets is calculated and divided by the sum of the number of pixels in each set. The intersection is scaled by a factor of two to allow the coefficient to range from 0 to 1. This coefficient then represents the percentage of pixels in the segmented output, which overlap with the ground truth data. A score of one indicates perfect agreement with the ground truth label while a score of zero indicates that no points of the segmented image were correctly classified.

In order to segment images with more than two classes, new methods must be developed to handle the increase in classifications. Due to the availability of multiple image modalities for each patient scan, segmentation networks for medical MRI scans can take advantage of this additional information in one of two approaches. The first is to include each modality as an additional dimension in the image input, acting in the same manner that color channels would for a standard photographic image. This has the side effect of simplifying the number of networks

and training needed while increasing the memory and computational requirements of the neural network.

The second approach is to train separate neural network models for each desired segmentation label. This would result in a single network for a single segmentation label. The networks would then act as an ensemble to layer the final outputs and create a multi-label segmented image output. This method can also take advantage of previously known features of the different imaging modalities. A network developed by Wang et al. uses this approach and additionally trains the network separately on each of the axial, sagittal, and coronal perspectives [13, 14]. Implementing network design and training in this manner reduces the technical and training overhead of developing the network while also increasing the interpretability of the neural network by relying on standard practices for segmenting the image. However, the potential overlap of the available information in each modality is lost by separating the segmentation into unique networks, potentially limiting the performance of the network through a limitation of the available data.

## 2.5 Detailed Neural Network Techniques

### Batch Normalization

Batch normalization is an additional step at each layer of a deep learning model which seeks to reduce training time and improve performance by solving an issue with the statistical distribution of input data known as internal covariate shift [15]. When ordered data such as an image is used as input to the first layer of a deep neural network, the layer learns something from

the distribution of values in the image and passes that information along to the following layer. However, the data which is passed on does not necessarily have the same distribution as the previous layer due to non-linear functions common in deep neural network layers. Therefore, the following layer has to learn features based on a new statistical distribution. The issue arises during the training process as the parameters of the network are updated. These changing parameters cause the outputs of each following layer to change and thus also cause a change in the distribution from what the layer was previously learning. As the target distribution for each layer is constantly changing during the training process, the speed at which the model trains decreases [16].

The process of batch normalization is used to lessen this effect by normalizing the input mini-batch at each layer of the network. Normalization is calculated for each mini-batch and along each dimension as follows:

$$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

In the above equation, $\hat{x}$ is the normalized data point, $x$ is the original data point, $\mu$ is the mean of the data array, $\sigma$ is the variance of the data array, and $\epsilon$ is a small positive constant to provide stability in cases where the variance is zero or close to zero. This normalization process ensures that the distribution of data is more consistent throughout the training process. Additionally, the normalization of data has the added effect of regularizing the data and reducing the need for dropout layers [16].

Dropout

The dropout layer within each convolutional and deconvolutional block serves as an additional point of neural network regularization. The dropout layers have a probability hyper-parameter which governs their influence on the training process. During each mini-batch, a percentage of the nodes in the neural network are forced to a zero output based on the hyperparameter of the dropout layer. In the interpolation network model, this is set to 10%. By forcing some of the nodes in the network to zero, the contribution of those nodes towards reconstructing the in-between slice is ignored. This forces the network to generalize a small amount of the feature extraction across all of the nodes in the network. This has the result of making the network more resilient to overtraining on the input data.

Activation Function

The convolutional layers of the network alone are a series of linear functions. In order to allow the neural network to model non-linear functions accurately, introduction of nonlinear layers between each of the convolutional layers is necessary [17]. A common method of incorporating this into a network is to add nonlinear activation functions, also called transfer functions, between the hidden layers of the network. An example of one common activation function is the Rectified Linear Unit (ReLU) which is defined as the following:

$$f(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

The activation function at the output of each internal convolutional block was selected to be a LeakyReLU layer. The LeakyReLU function is a modification of the basic ReLU and is defined as follows:

$$f(x) = \begin{cases} x, & x > 0 \\ ax, & x \leq 0 \end{cases}$$

For this function, $a$ is a small positive constant, in the case of this paper, $a$=0.3.

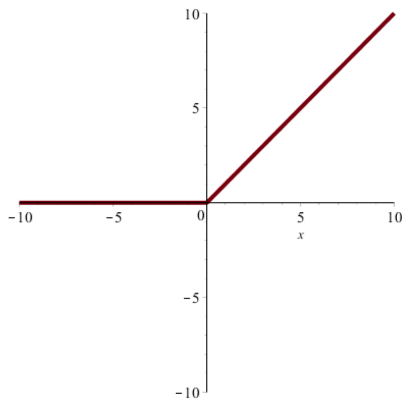



**Figure 6: ReLU Activation Function**

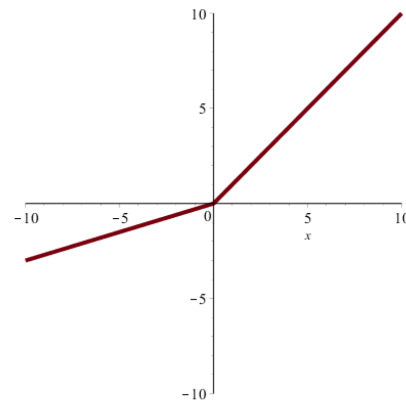



**Figure 5: LeakyReLU Actication Function**

Building upon the principals of the ReLU activation function, the LeakyReLU has a small positive gradient for input values below zero. The activation function was designed in this manner in order to protect against the "dying ReLU problem" [18]. The dying ReLU problem occurs when the output of a node using the ReLU activation function is negative, causing the gradient returned during back propagation to be zero indefinitely. This results in the neuron sticking in an off state since a gradient of zero will cause no change in the weights and biases feeding into the activation function. Maintaining a small positive slope for negative inputs allows nodes activated using LeakyReLU an opportunity to recover from this issue.

The output layer of the interpolation network simply uses a ReLU activation layer with an unsigned 15-bit input and output as this matches the range of values possible in a slice of the MRI volume. Because the network is generative, selecting an activation function that weights

possible pixel values evenly based on input and spans the full range of necessary outputs simplifies the reconstruction process as the output and ground truth slices can be compared directly without transformations or mapping functions.

## 2.6 Reason for Selecting the Encoder-Decoder Structure for the Neural Network

The reason for selecting the structure of an encoder-decoder with skip connections as the general structure for the slice interpolation neural network was due to several inherent qualities of this architecture.

The first supportive quality is that the encoder-decoder structure is a generative network designed to reconstruct a set of data based on a lower-dimensional set of features. MRI volumes are a good candidate for this form of compression and feature-based reconstruction because the image volumes all contain the same type of content. In this case the, brains of patients undergoing treatment for brain tumors. The encoder-decoder can identify common structures within the image slices and describe them with a smaller number of latent features than the original set of pixel data.

The network utilizes the features found by the decoder in a reconstruction of the input, which is cognizant of the common structure of a brain MRI volume. The decoder learns unique convolutional masks and biases which rebuild the input image from the latent features.

Finally, the skip connections of this architecture enable the network to use features extracted earlier in the encoder half of the network to aid in reconstructing the input. Applying additional mathematical functions to the skip connection data, such as averaging the pair of input

slices as they move through the skip connection, allows the decoder to work with dense features in the same dimensions as the eventual output slice.

A significant drawback to this architecture is the memory required to store and process the model during training. With each downsampling stage in the network, the number of convolutional filters need to double in order to minimize the amount of information lost in the process. Deep networks also need to be built to extract a high level, abstract feature map of the underlying data. Therefore, the number of convolutional filters combined with the requisite number of stages for a deep network quickly inflates the number of trainable parameters in the network. Likewise, the number of input slices contributes a multiplicative effect to the memory requirements for the encoder half of the network.

# 3. PROPOSAL FOR INVESTIGATING SLICE INTERPOLATION

As mentioned in Section 1.2, the resolution between slices is greater than the resolution of pixels in the slice itself. This results in less availability of information along the depth axis of the image volume. I conducted an exploration of the impact of this lower slice resolution on the accuracy of medical imaging analysis algorithms, such as image segmentation. The experiment involves down-sampling the number of slices available in an image volume for MRI scans of patients with gliomas and reconstructing the lost slices through various methods. Reconstruction methods include a simple linear interpolation between neighboring slices as the baseline performance method, and a new proposal of a U-net based CNN to reconstruct the missing slice given the neighboring slices.

This is accomplished by modifying the general U-net architecture to utilize a two image input and one image output setup. The network receives the pair of input slices as a two-channel image, with the first channel indicating the bottom slice and the second channel the top slice. At some point in the network, these channels must be merged so that a single channel image slice is given as the output of the network. This is accomplished by averaging the channels together at each point where information transfers from the encoder half of the network to the decoder half. Therefore, the channels are averaged together into a single-channel matrix at each of the skip connections and in the transition from the bottleneck layer to the first deconvolution block of the decoder.

A similar area of research to the proposed slice interpolation issue has been explored by many others for the purpose of interpolating between frames of digital video to achieve digital video data compression and picture quality enhancement. Video frames capture motion across time, whereas typical MRI slices strictly capture changes across a volume space. Therefore,

current methods used to interpolate between frames of digital video may be applicable to an MRI scan as the format of the data is similar. However, the content reflected by changes in the frame or slice depth (time vs. volume location) reflects a different set of learnable features within the data.

A method of video frame interpolation using CNNs called Deep Voxel Flow was proposed by a team from the University of Illinois at Urbana-Champaign and Google Research which used a general form of U-net in tandem with a technique of following motion between images known as optical flow [19]. Deep Voxel Flow utilizes the Encoder-Decoder structure to produce a 3D voxel flow field from the pair of input images. Then, by assuming the data in the voxel flow field is locally linear, they were able to resample using trilinear interpolation between the frames to create an estimation of the restored video frame.

### 3.1 Data Set Selection and Preprocessing

The dataset used for this research is the BraTS 2018 dataset [20, 21, 22]. This dataset includes the scans from 285 patients collected from 2012 to 2013 and labeled in 2017. The image volumes were gathered from 19 different institutions and resampled to a voxel size of 1 mm x 1 mm x 1 mm. The skulls had been removed from the image volumes in the pre-processing stage before the dataset being made publicly available. In addition, all of the images were registered to the same anatomical template to ensure that the images could be related under a shared coordinate system.

The pre-processing actions performed in the completion of this thesis were conducted to preserve the integrity of the information stored within the image. In order to down-sample along

the slice axis, odd-numbered slices, using a 0-based indexing system, were discarded for each of the four modalities and the resulting compressed image volume saved as a separate dataset. The purpose of discarding the odd-numbered slices was to ensure that the first and last image slices were preserved to ensure that the restored image volume would match the original dimensions of the original data.

Additionally, in any case where the width or height of the image slice needed to be reduced, such as for performance purposes during testing, a crop function was performed instead of a zoom or resampling function. Using a crop preserves the physical dimensions of the image volume and does not alter any of the remaining data, such as would be done in a linear or nearest-neighbor interpolation.

All image volumes are stored in the NIfTI format specified by the National Institute of Health. Image loading and saving were performed using the Python library SimpleITK. Matrix operations on the loaded image volumes were performed using NumPy methods and matrix arithmetic.

## 4. INTERPOLATION NEURAL NETWORK ARCHITECTURE

My attempt to improve the reconstruction of the intermediary MRI slice is performed using a convolutional encoder-decoder neural network. This network follows a similar architecture to U-net by employing several convolution and max-pooling layers to reduce the dimension in three steps from 240 by 240 pixels with 16 convolutional filters to 120 by 120 with 32 filters, 60 x 60 with 64 filters, 30 x 30 with 128 filters, and finally 15 x 15 with 256 filters. This process is then mirrored with the corresponding deconvolution and upsampling layers to return to the original 240 x 240 slice resolution. Two slices, the paired top and bottom slices, are fed into the network together. The same filters are applied to each slice at each layer. However, after each downsampling layer, the output feature map pair for the slices are averaged into a single feature map and, using a skip connection, are concatenated to the corresponding resolution deconvolution layer. The resulting overall architecture takes the same general shape as U-net. The model architecture is shown in Figure 7 to demonstrate how averaging the pair of input slices across the skip connections is modeled in the architecture. Both slices could be subject to independently valued convolution filters, however, fixing the filters to apply identically to both input slices greatly reduces the memory requirement and enables the model to be efficiently trained on current generation hardware.

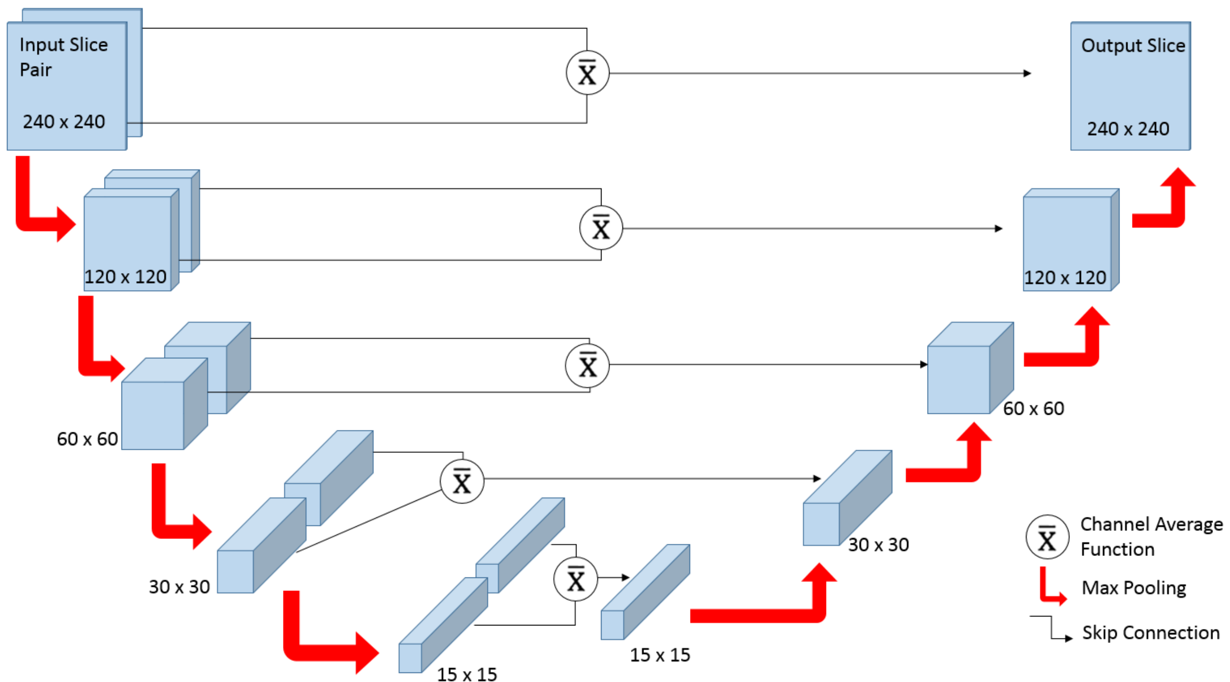



**Figure 7: Example layout of interpolation model architecture with channel averaging residual skip connections**

In each convolutional block is a dropout layer set to a dropout probability of 10%, a batch normalization layer, and a Leaky ReLU as the activation layer. The loss function for training is the Mean Square Error (MSE) and the optimizer is Adamax with a learning rate of 0.0001.

Batch normalization is an included layer in each convolutional and deconvolutional block of this encoder-decoder model. The purpose of this layer is to normalize the input data along each mini-batch to prevent or lessen the impact of internal covariate shift. The issue of internal covariate shift is caused by the differences in data distribution between inputs. This is expressed in the MRI volumes by the large fluctuations in pixel intensity throughout each slice. Some slices, especially those in the beginning and the end of the volume have a constant zero value, whereas the maximum pixel value in other slices ranges from 1 to the maximum value of an unsigned 15-bit integer of 32,767 as shown in Figure 8:

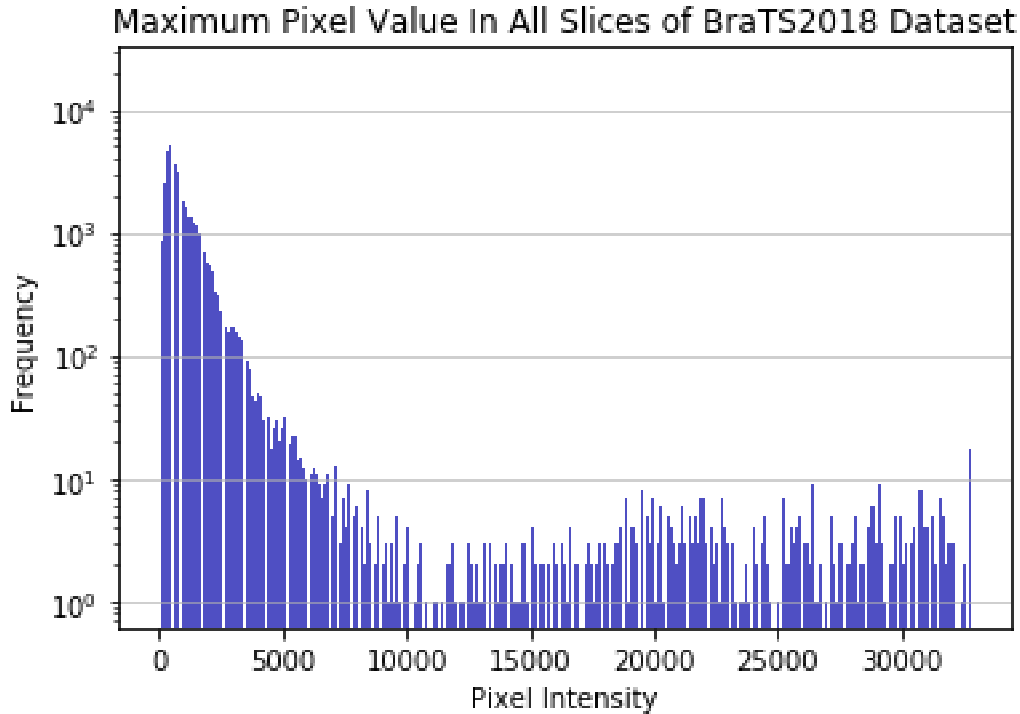



**Figure 8: Histogram of maximum pixel intensities in the BraTS dataset with a log y-axis and linear x-axis**

Figure 8 is scaled logarithmically along the y-axis to demonstrate that although the higher valued pixels are not especially common in slices, there is a wide range of maximum values. Therefore, using my chosen mini-batch size of 4, it is possible that a group of slices may be dominated by one slice containing these high-intensity pixels. This outlier slice would then shift the representative distribution of pixel data disproportionately toward itself. Normalizing across batches lessens the impact of this outlier behavior and thus allows the network to train towards a more generalizable latent representation of the underlying data features.

The 10% parameter for dropout is lower than the 20% from the video frame interpolator in X. Mao's image restoration encoder-decoder [23]. This lower hyperparameter was selected due to the consistent nature of the MRI dataset. Since each MRI volume is viewed from the same

perspective, centered within the volume, and registered to a common brain atlas, there is less need to regularize based on the size and location of the information and, therefore, a small amount of dropout results in a similar impact.

## 4.1 Interpolation Network Experimental Configurations

Depending on the version of interpolation network being tested, the network contains either 500,000 or 7,500,000 trainable parameters. The difference in trainable parameters arises from altering the number of convolutional filters in each block with the smallest network containing one quarter the number of filters as the largest network. A third version of the network was tested in which the bottleneck was doubled in size to 30 x 30, an additional convolutional block was added without downsampling, and the number of convolutional filters was half of the amount in the largest network described previously. These changes resulted in a model that also has 7,500,000 trainable parameters.

The presented configurations represent the models that were most stable during smoke tests and short training sessions on a reduced sized dataset. Other configurations were tested but were not selected for a full training session due to unstable training loss, slow training speed regardless of the learning rate, or no convincing signs of potential improvement in the training set.

These smaller experimental configurations were executed on a small reduced training set of 15 MRI volumes. The experimental model changes include using the Adam optimizer instead of Adamax, increasing the dropout probability hyperparameter to 20%, removing the skip connection from the first convolutional block, removing batch normalization, using average

pooling instead of max pooling, and using a stride on the convolution filters as opposed to pooling of any sort for the downsampling step. The changes in the optimizer, the inclusion of batch normalization, and pooling or stride type made no discernable difference or a small loss in performance on the training set and, as such, did not pass screening to be utilized in the 24 hour or greater full dataset tests. Removing dropout altogether made training less stable and decreased the speed at which the network trained, similar to how a network with too high of a learning rate performs. Removing the first skip connection drastically reduced performance and failed to reconstruct the intermediary slices even when over-trained on a single MRI volume in an additional test. Each of the described modifications were not chosen to be tested further not only based upon poor or negligible performance, but additionally due to the techniques not aligning with standard modern practices in model design. The examples of such modern practices were drawn from the literature describing and introducing Deep Voxel Flow, the BRATS 2018 segmentation network by A. Myronenko, and the image restoration encoder-decoder by X. Mao et al., which all contain some form of batch normalization, dropout, first block skip connections, max pooling, and Adamax [19, 23-24].

All final model training and inference tasks were performed using a local Nvidia RTX 2080S GPU with 8GB of GDDR6 VRAM with a 5% overclock on core and memory speeds. The dataset was additionally stored on a local solid-state drive. Initial segmentation and interpolation model experiments were performed on an Nvidia P100 GPU with 16 GB of GDDR5 VRAM located on a pre-emptible Google Cloud instance. The formerly described local machine was chosen as the final training solution due to faster training speed, higher training stability, faster disk access times, and lower long term costs. This choice of computing solution came at the cost of lower memory and thus constrained the maximum possible model size to 8GB.

The network on the upper limit of trainable parameters required training of 36 hours a network before the validation loss stabilized around a nearly constant value. The smaller network of 500,000 parameters only trained for 24 hours before a similar stable loss on the validation set was achieved.

However, despite the hit to accuracy, the smaller network was much quicker to train and use. The overall compressed model weights and bias storage requirement of 9.0 MB vs 93 MB for the larger network using the HDF5 storage format.

## 5. INTERPOLATION RESULTS

The trained interpolation models were tested against the MRI slices from 105 patient image volumes across all imaging modalities. This test set of 105 patients is independent of the training and validation sets. Image slice pairs were predicted by the model in batches of 20. This is an increase from the batch size of 4 used in training. Since predicting an output using a trained neural network requires less memory than the training phase, the batch size was increased to increase prediction throughput several-fold. The average prediction speed for each interpolation was 10 milliseconds, independent of the model configuration.

The metric for evaluating the final performance of each interpolation method is the peak signal-to-noise ratio (PSNR) measured in decibels (dB). The PSNR is calculated as follows:

$$PSNR = 20 \cdot \log_{10}\left(\frac{MAX_I}{\sqrt{MSE}}\right)$$

In this case, MSE is the mean square error of the entire interpolated image slice and $MAX_I$ is the maximum image pixel value. In order to allow comparability with the more common unsigned 16-bit data type, all images were scaled to fit in a 16-bit format by multiplying the unsigned 15-bit images by 2. This makes $MAX_I$ in this study equal to 65,535. The PSNR image quality metric attempts to quantify how much of the original image slice information is retained after noise and data loss is introduced by the reconstruction algorithm. The signal is the original intermediary slice and the noise is the image artifacts and data corruption due to the interpolation. An image pair with an MSE of just 1 gives a PSNR of 96 dB and an image pair in which each pixel has the maximum possible error of 65,535 gives a resulting PSNR of 0 dB.

The average PSNR for the standard model, the reduced parameter model, and the wide bottleneck model are given as Models A, B, and C respectively, and compared against linear interpolation in Table 1:

**Table 1: Slice Reconstruction PSNR of Tested Interpolation Methods**

| Interpolation Method | PSNR (dB) |
| --- | --- |
| Model A | 65.28 |
| Model B | 62.88 |
| Model C | 65.33 |
| Linear Interpolation | 65.97 |

These results for all models initially indicate performance slightly worse than the linear interpolation. The markedly worse performance of Model B can be attributed to its inability to learn and retain enough features from the dataset. Model B differs from Model A by having a narrower bottleneck layer and fewer convolutional filters at each layer. Model B differs from Model C by containing one fewer convolutional block and half the number of convolutional filters. Therefore, Model B is constructed in such a way as to have an order of magnitude fewer trainable parameters than Models A and C. Given that all three were trained on the same data with the same data randomizing seed, we can conclude that Model B was not able to extract enough features from the data to enable an accurate reconstruction when compared to the other models and the linear interpolation method.

Model A and C were roughly equivalent in performance with only a 0.05 dB difference in their average PSNR. However, they were both over half a decibel below the performance of the

linear interpolation method indicating a small but clear shortfall of the baseline interpolation method.

One point of concern regarding the result of the encoder-decoder output is that it does not seem that the neural network models are learning a latent representation of the brain structure in order to perform the interpolation. When comparing results visually, the errors of the network are extremely similar to those errors produced by linear interpolation, namely that discontinuities in the brain shape are not filtered or reduced in size, but instead simply average to an intermediate grey value. This is shown by the two dots in the top left and right of the brain stem in the slices depicted in Figure 9:

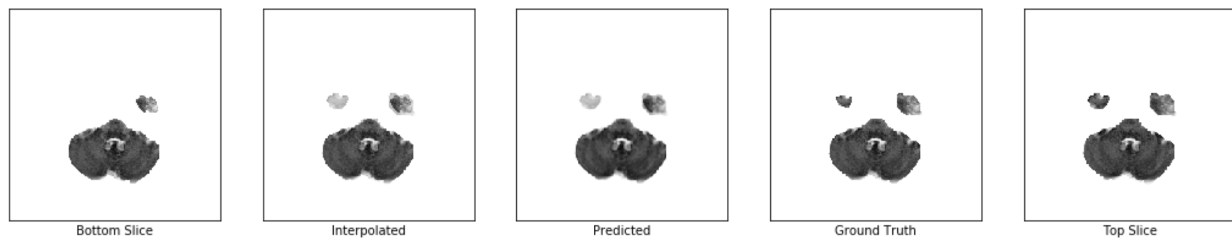



**Figure 9: MRI Volume slices showing, from left to right, the first input slice, the linear interpolated result, the neural network predicted intermediary slice, the ground truth intermediary slice, and the second input slice.**

Figure 9 shows an early slice in the image volume, along the base of the skull where the brain stem leads into the middle of the head. Going from the bottom slice in the image in the direction of the top slice, the front of the brain behind the eyes begins to enter the view as an overhanging mass, with the very bottom tips of it represented by the two dots which appear in the Top Slice of Figure 9. The Ground Truth image is the true slice between the bottom and top input slices, the predicted image is the slice generated by the neural network, and the interpolated image is the slice generated by simply averaging the top and bottom slices.

In order to investigate how the neural network model is interpolating the image slices, two further experiments were conducted in which the slices used as input were not directly adjacent to the interpolated slice, but further out. The past experiment used a distance between input slices of 1. The next two experiments used a distance of 3 and 5 slices, respectively. The goal of each experiment is still to interpolate only the center slice between the pair of input MRI slices. These increased slice distance experiments were performed using Models A and C. The reduced size interpolation configuration of Model B was not selected as the performance was significantly poorer than the other model configurations. The results of Models A and C in this experiment are presented in Table 2.

**Table 2: PSNR of Interpolation Methods at Varying Input Slice Distances**

| Interpolation Method | 1-Slice Distance PSNR (dB) | 3-Slice Distance PSNR (dB) | 5-Slice Distance PSNR (dB) |
|---|---|---|---|
| Model A | 65.28 | 61.34 | 59.64 |
| Model C | 65.33 | 61.45 | 59.83 |
| Linear Interpolation | 65.97 | 61.11 | 59.39 |

The results of this experiment show a slightly higher performance of the neural network interpolator over the linear averaging method. Overall interpolation performance decreases significantly as the slice distance grows. Despite this, the neural network interpolators increase in interpolation performance relative to the linear method accordingly to the progression of the trials. The neural networks perform relative to the linear method by an average of -0.665 dB at a slice distance of 1, +0.285 dB at a slice distance of 3, and +0.345 dB at a slice distance of 5.

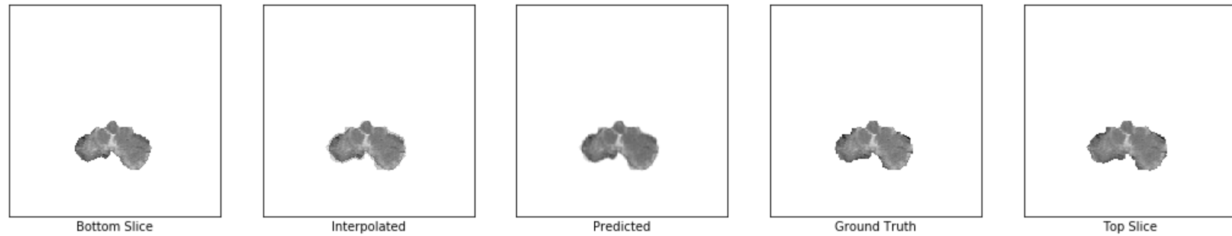



**Figure 10: Sample results of Model C at 1-Slice Distance. MRI Volume slices showing, from left to right, the first input slice, the linear interpolated result, the neural network predicted intermediary slice, the ground truth intermediary slice, and the second input slice.**

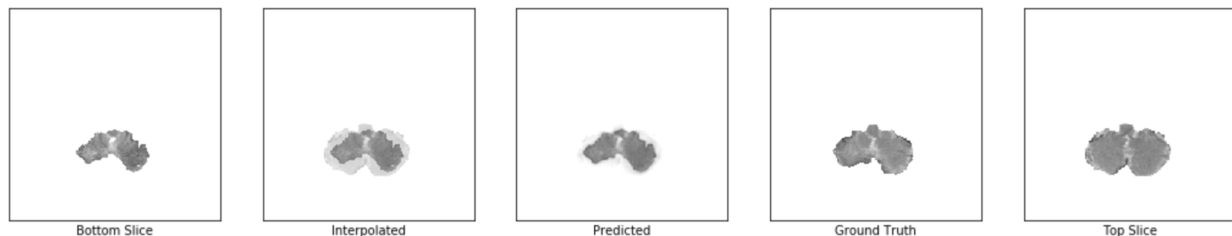



**Figure 11: Sample results of Model C at 5-Slice Distance. MRI Volume slices showing, from left to right, the first input slice, the linear interpolated result, the neural network predicted intermediary slice, the ground truth intermediary slice, and the second input slice.**

The image slices shown in Figures 10 and 11 are centered on the same middle slice. This is to demonstrate the differences in reconstruction between the different slice distances given the same target. In particular, there is a small visual difference between all three middle slices in Figure 10. However, in Figure 11 the linear interpolated slice has distinct ringing around the core of the image. This ringing effect is significantly muted in the neural network reconstructed image slice.

Intuitively these results seem reasonable due to how each method is performing the interpolation. The linear averaging method works best when both input slices are quite similar, which occurs when the distance between the slices is small. As the distance between the input

slices increases, the similarities between the contents of the input slices decreases and methods which takes into account shared features in brain structure should improve in relative performance.

## 5.1 Segmentation Results of the Interpolated MRI Volumes

I implemented the segmentation model introduced by Wang, Li, Ourselin, and Vercauteren developed for the BraTS 2017 challenge and modified it to operate with the BraTS 2018 data. In an attempt to evaluate how this model performs on the original data compared to the interpolated MRI volumes, the model was run on the complete 2018 dataset of 285 image volumes and calculated the Dice Coefficients for the original data, the data reconstructed by linear interpolation, and the volumes reconstructed using Model C. The segmentation results for each interpolation method are presented in Table 3.

**Table 3: Dice Coefficient for Tumor Segmentation of Processed Datasets**

| | Dice Coefficient | | | Standard Deviation | | |
|---|---|---|---|---|---|---|
| Experiment | ET | ED | NCR/NET | ET | ED | NCR/NET |
| Original Data | 0.718 | 0.800 | 0.784 | 0.197 | 0.132 | 0.242 |
| Linear Interpolation | 0.708 | 0.797 | 0.786 | 0.202 | 0.130 | 0.233 |
| Neural Network Interpolation | 0.707 | 0.799 | 0.782 | 0.205 | 0.135 | 0.224 |

The results of this experiment show that the loss of information throughout the reconstruction process does not significantly impact the performance of this model. Scores for each were well within one standard deviation of the original data segmentation.

A likely cause of this stable performance is due to Wang et al.'s implementation of the segmentation model as a cascading ensemble of convolutional models. Rather than segmenting from a single volume or single anatomical perspective, the creator of this segmentation model chose to implement 9 separate models. These 9 models are trained separately on the three anatomical axes for each targeted segmentation class. For instance, there is a specific model for MRI slices obtained in the coronal view and producing segmentation of the tumor core. Each output of the model trios for each segmentation class are averaged and then contribute their findings to a final deep learning model which produces the complete segmentation. This multi-perspective ensemble approach likely lends the model resistance to information degradation along a single axis, such as that produced by my interpolation experiments.

# 6. POTENTIAL ISSUES WITH PROPOSED INTERPOLATION APPROACH

The proposed method of reconstructing intermediary MRI slices using convolutional neural networks is based on several assumptions that may have limited the potential performance of this approach. These assumptions consist of the overall encoder-decoder design, the practical implementation of the model, and the dataset used for training.

The primary goal of using the encoder-decoder architecture was to investigate if a simple interpolation technique, such as linear interpolation, could be improved by using an algorithm that takes into consideration the local features of images with a shared topic. In this case, a convolutional encoder-decoder will be able to account for local features in the images, while the shared topic is a well-studied dataset of MRI volumes. Therefore, the neural network should be capable of learning an abstract representation of the pixel features and use this representation to reconstruct a compressed volume of MRI slices. This is opposed to an approach like linear interpolation which ignores the relationship between neighboring pixels and thus cannot take advantage of features in a dataset with a common topic. However, due to the semi-supervised nature of training an encoder-decoder neural network, regression of the model towards linear methods is always a possibility as higher-performing representations are not guaranteed to exist. This is because a non-linear neural network, such as the proposed encoder-decoder architecture, is guaranteed to have poor local minima [25]. In addition, due to the large number of parameters in all of the researched model configurations and that only a few parameters are needed to mimic a linear averaging of MRI slices, there must exist at least one local minimum in the training loss function in which the model approximates linear interpolation of the input frames. This ability for a neural network to approximate any function is known as the Universal Approximation Theorem and has been proven to extend to deep CNNs [26]. The conclusion regarding the

assumed architecture then is that a model which ignores the abstract features of the subject and instead utilizes a nearly linear interpolation method is always a possible outcome of the model training process.

The potential for the network to arrive at the nearly trivial linear interpolation solution is further realized by the existence of the skip connections. The skip connections allow features from shallow layers to be used in the reconstruction taking place in the decoder section of the model. The features of the shallow layers have not yet been parsed through the full range of convolutional layers and thus may represent a subset of features which are similar to the original input. This quality of skip connections is useful in recovering lost spatial information, however, the novel method introduced in this thesis of averaging together input features along the skip connection may reinforce the tendency of the network to reproduce a linear method of interpolation.

The dataset used for training and validation consisted of 41,000 pairs of MRI volume slices taken from 285 patients. The main point of concern is the number of patients included in this dataset compared to the overall size of datasets used to train similar neural network architectures in domains outside of healthcare. For instance, the authors of the Deep Voxel Flow video frame synthesis model trained their model on the UCF101 dataset which consists of 13,000 videos. At the stated 25 frames per second, this video dataset consists of 2.4 million frame pairs. Datasets of comparable size in the medical field are extraordinarily rare [27]. With fewer examples of key spatial features available, these smaller healthcare datasets may not contain enough data to accommodate the architectures of generalized encoder models developed and proven on much larger datasets.

# 7. CONCLUSION

Acquiring and analyzing medical images is a critical step in working with difficult to treat diseases such as brain tumors. Modern techniques for automated segmentation of brain tumors have increased dramatically and may soon see use in a clinical setting. Among these techniques, we examined the construction and implementation of convolutional neural networks for segmentation tasks. These neural network models have been incrementally improved over the last decade due to breakthroughs in model training methodology and model architecture. In this thesis, we explored how these breakthrough technologies in segmentation models can be applied to generative models of neural networks within the same data topic.

The generative model introduced aimed to interpolate intermediate slices from an MRI volume along a single axis and in the process compare the generated slices against a linear method. The generative neural network, based on a modification of the U-net segmentation model, incorporated newer modeling techniques such as batch normalization, LeakyReLU activation functions, and dropout.

Using the peak signal-to-noise ratio metric, the performance of the interpolation neural network did not exceed that of the linear method. However, further experiments using an exaggerated form of the initial trials in which the slice interpolation occurs over an extended distance provided some indication that the neural network interpolation method is capable of using intrinsic features of the dataset to reconstruct an intermediary slice.

Although the performance of the interpolating neural network is not yet superior to the linear interpolation method, the experiments do indicate that linear averaging of MRI slices does not stand as the upper limit of interpolation performance in this imaging domain. Also, as the

initial inter-slice resolution decreases, the performance of a neural network-based solution improves relative to linear interpolation.

## 8. FUTURE IMPROVEMENTS

Due to the large processing requirements of deep learning, not all considered improvements and alterations to the experiments could be examined during the development process. Therefore, I will provide several avenues for further research which could be explored given additional time and computing resources, or advancements in computing technology.

A considerably more complex but potentially fruitful approach to improving the interpolation ability of the neural network model is to expand the number of input slices. Rather than using only the adjacent slices, additional pairs of slices from deeper in the MRI volume could be included in the encoder stage of the model. However, the drawback to this approach is that adding only a single extra pair of slices doubles the memory footprint of the encoder half of the network. Additionally, a linear interpolation is no longer an appropriate baseline as B-spline interpolation is a proven efficient method for interpolating between multiple MRI slices [28].

The dataset used for this research is significantly smaller than would be found in a non-healthcare commercial setting where neural networks have seen greater success in engineering applications. Therefore, a simpler improvement would be to augment the dataset with either transformed image slice pairs or introducing synthetic data. Examples of common transformations are flipping and rotating random images along an axis, performing image rotations, and introducing an intensity shift to the image. Rotating or flipping image slices along any axis that is not aligned with the edges of the volume is not advisable due to the necessary interpolation or averaging of pixels to achieve rotation angles such as 30° or 60°. The dataset can also be expanded through the use of automatically generated synthetic data to allow for a more

diverse dataset, which still contains accurate ground truth labels. A method for generating synthetic MRI volumes of glioblastoma occurrences was developed and implemented using a computational network [29].

In addition to dataset augmentation, transfer learning from model optimized on a large dataset could be utilized to lower the number of parameters that need to be trained on the smaller healthcare dataset. Video data would be an intuitive starting point to pre-train the model, however, one concern is that the information and objects in video frames represent changes in time and space. A purely spatial dataset would be preferable, such as geological survey data, in which learned spatial features may translate more appropriately to a similarly volumetric MRI dataset.

# LIST OF REFERENCES


[1] A. D'Alessio, G. Proietti, G. Sica, and B. M. Scicchitano, "Pathological and Molecular Features of Glioblastoma and Its Peritumoral Tissue," *Cancers*, vol. 11, no. 4, p. 469, Apr. 2019, doi: 10.3390/cancers11040469.

[2] "What Makes a Brain Tumor High-Grade or Low-Grade?," *Dana-Farber Cancer Institute*, Apr. 03, 2018. https://blog.dana-farber.org/insight/2018/04/makes-brain-tumor-high-grade-low-grade/ (accessed May 20, 2020).

[3] M. C. Mabray, R. F. Barajas, and S. Cha, "Modern Brain Tumor Imaging," *Brain Tumor Res Treat*, vol. 3, no. 1, p. 8, 2015, doi: 10.14791/btrt.2015.3.1.8.

[4] "Brain MRI." https://www.cedars-sinai.edu/Patients/Programs-and-Services/Imaging-Center/For-Patients/Exams-by-Procedure/MRI/MRI-Brain.aspx (accessed May 20, 2020).

[5] E. V. Reeth, I. W. K. Tham, C. H. Tan, and C. L. Poh, "Super-resolution in magnetic resonance imaging: A review," *Concepts in Magnetic Resonance Part A*, vol. 40A, no. 6, pp. 306–325, 2012, doi: 10.1002/cmr.a.21249.

[6] A. W. Toga and P. M. Thompson, "The role of image registration in brain mapping," Image Vis Comput, vol. 19, no. 1–2, pp. 3–24, Jan. 2001, doi: 10.1016/S0262-8856(00)00055-X.

[7] "(PDF) Macroscopic Cerebral Tumor Growth Modelling from Medical Images: A Review," ResearchGate. https://www.researchgate.net/publication/325309271_Macroscopic_Cerebral_Tumor_Growth_Modelling_from_Medical_Images_A_Review (accessed May 31, 2020).

[8] J. E. Villanueva-Meyer, M. C. Mabray, and S. Cha, "Current Clinical Brain Tumor Imaging," *Neurosurgery*, vol. 81, no. 3, pp. 397–415, Sep. 2017, doi: 10.1093/neuros/nyx103.

[9] S. Ruder, "An overview of gradient descent optimization algorithms," arXiv:1609.04747 [cs], Jun. 2017, Accessed: May 31, 2020. [Online]. Available: http://arxiv.org/abs/1609.04747.

[10] A. Wadhwa, A. Bhardwaj, and V. Singh Verma, "A review on brain tumor segmentation of MRI images," Magnetic Resonance Imaging, vol. 61, pp. 247–259, Sep. 2019, doi: 10.1016/j.mri.2019.05.043.

[11] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," *arXiv:1505.04597 [cs]*, May 2015, Accessed: May 20, 2020. [Online]. Available: http://arxiv.org/abs/1505.04597.

[12] C. H. Sudre, W. Li, T. Vercauteren, S. Ourselin, and M. J. Cardoso, "Generalised Dice overlap as a deep learning loss function for highly unbalanced segmentations," arXiv:1707.03237 [cs], vol. 10553, pp. 240–248, 2017, doi: 10.1007/978-3-319-67558-9_28.

[13] Guotai Wang, Wenqi Li, Sebastien Ourselin, Tom Vercauteren. "Automatic Brain Tumor Segmentation using Cascaded Anisotropic Convolutional Neural Networks." In Brainlesion: Glioma, Multiple Sclerosis, Stroke and Traumatic Brain Injuries. Pages 179-190. Springer, 2018. https://arxiv.org/abs/1709.00382

[14] Eli Gibson, Wenqi Li, Carole Sudre, Lucas Fidon, Dzhoshkun I. Shakir, Guotai Wang, Zach Eaton-Rosen, Robert Gray, Tom Doel, Yipeng Hu, Tom Whyntie, Parashkev Nachev, Marc Modat, Dean C. Barratt, Sébastien Ourselin, M. Jorge Cardoso, Tom Vercauteren. "NiftyNet: a deep-learning platform for medical imaging." Computer Methods and Programs in Biomedicine, 158 (2018): 113-122. https://arxiv.org/pdf/1709.03485

[15] N. Bjorck, C. P. Gomes, B. Selman, and K. Q. Weinberger, "Understanding Batch Normalization," in Advances in Neural Information Processing Systems 31, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 7694–7705.

[16] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *arXiv:1502.03167 [cs]*, Mar. 2015, Accessed: May 20, 2020. [Online]. Available: http://arxiv.org/abs/1502.03167.

[17] F. Agostinelli, M. Hoffman, P. Sadowski, and P. Baldi, "Learning Activation Functions to Improve Deep Neural Networks," *arXiv:1412.6830 [cs, stat]*, Apr. 2015, Accessed: May 20, 2020. [Online]. Available: http://arxiv.org/abs/1412.6830.

[18] L. Lu, Y. Shin, Y. Su, and G. E. Karniadakis, "Dying ReLU and Initialization: Theory and Numerical Examples," *arXiv:1903.06733 [cs, math, stat]*, Nov. 2019, Accessed: May 20, 2020. [Online]. Available: http://arxiv.org/abs/1903.06733.

[19] Z. Liu, R. A. Yeh, X. Tang, Y. Liu, and A. Agarwala, "Video Frame Synthesis using Deep Voxel Flow," *arXiv:1702.02463 [cs]*, Aug. 2017, Accessed: May 20, 2020. [Online]. Available: http://arxiv.org/abs/1702.02463.

[20] B. H. Menze, A. Jakab, S. Bauer, J. Kalpathy-Cramer, K. Farahani, J. Kirby, et al. "The Multimodal Brain Tumor Image Segmentation Benchmark (BRATS)", IEEE Transactions on Medical Imaging 34(10), 1993-2024 (2015) DOI: 10.1109/TMI.2014.2377694

[21] S. Bakas, H. Akbari, A. Sotiras, M. Bilello, M. Rozycki, J.S. Kirby, et al., "Advancing The Cancer Genome Atlas glioma MRI collections with expert segmentation labels

and radiomic features", Nature Scientific Data, 4:170117 (2017) DOI: 10.1038/sdata.2017.117

[22] S. Bakas, M. Reyes, A. Jakab, S. Bauer, M. Rempfler, A. Crimi, et al., "Identifying the Best Machine Learning Algorithms for Brain Tumor Segmentation, Progression Assessment, and Overall Survival Prediction in the BRATS Challenge", arXiv preprint arXiv:1811.02629 (2018)

[23] X. Mao, C. Shen, and Y.-B. Yang, "Image Restoration Using Very Deep Convolutional Encoder-Decoder Networks with Symmetric Skip Connections," in Advances in Neural Information Processing Systems 29, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds. Curran Associates, Inc., 2016, pp. 2802–2810.

[24] A. Myronenko, "3D MRI Brain Tumor Segmentation Using Autoencoder Regularization," in Brainlesion: Glioma, Multiple Sclerosis, Stroke and Traumatic Brain Injuries, Cham, 2019, pp. 311–320, doi: 10.1007/978-3-030-11726-9_28.

[25] K. Kawaguchi, "Deep Learning without Poor Local Minima," *arXiv:1605.07110 [cs, math, stat]*, Dec. 2016, Accessed: May 20, 2020. [Online]. Available: http://arxiv.org/abs/1605.07110.

[26] D.-X. Zhou, "Universality of deep convolutional neural networks," Applied and Computational Harmonic Analysis, vol. 48, no. 2, pp. 787–794, Mar. 2020, doi: 10.1016/j.acha.2019.06.004.

[27] M. D. Kohli, R. M. Summers, and J. R. Geis, "Medical Image Data and Datasets in the Era of Machine Learning—Whitepaper from the 2016 C-MIMI Meeting Dataset Session," J Digit Imaging, vol. 30, no. 4, pp. 392–399, Aug. 2017, doi: 10.1007/s10278-017-9976-3.

[28] W. van Valenberg, S. Klein, F. M. Vos, K. Koolstra, L. J. van Vliet, and D. H. J. Poot, "An Efficient Method for Multi-Parameter Mapping in Quantitative MRI using B-Spline Interpolation," *arXiv:1911.07785 [physics]*, Nov. 2019, Accessed: May 20, 2020. [Online]. Available: http://arxiv.org/abs/1911.07785.

[29] L. Lindner, B. Pfarrkirchner, C. Gsaxner, D. Schmalstieg, and J. Egger, "TuMore: generation of synthetic brain tumor MRI data for deep learning based segmentation approaches," vol. 0579, p. 105791C, Mar. 2018, doi: 10.1117/12.2315704.

# APPENDICES

## Appendix A: Neural Network Model A Code

```
1.  import tensorflow.keras
2.  from tensorflow.keras import backend as K
3.  import numpy as np
4.  import os
5.
6.
7.  # # Model
8.
9.
10. import tensorflow.keras.layers as layers
11. import tensorflow.keras.models as models
12. from tensorflow.keras.initializers import orthogonal
13.
14.
15. def Conv3DLayer(x, filters, kernel, strides, padding, block_id, kernel_init=orthogonal(
    )):
16.     prefix = f'block_{block_id}_'
17.     x = layers.Conv3D(filters, kernel_size=kernel, strides=1, padding=padding,
18.                       kernel_initializer=kernel_init, name=prefix+'conv')(x)
19.     x = layers.LeakyReLU(name=prefix+'lrelu')(x)
20.     x = layers.MaxPooling3D(pool_size=(strides[0], strides[0], 1))(x)
21.     x = layers.Dropout(0.1, name=prefix+'drop')((x))
22.     x = layers.BatchNormalization(name=prefix+'conv_bn')(x)
23.     return x
24.
25. def Conv2DLayer(x, filters, kernel, strides, padding, block_id, kernel_init=orthogonal(
    )):
26.     prefix = f'block_{block_id}_'
27.     x = layers.Conv2D(filters, kernel_size=kernel, strides=1, padding=padding,
28.                       kernel_initializer=kernel_init, name=prefix+'conv')(x)
29.     x = layers.LeakyReLU(name=prefix+'lrelu')(x)
30.     x = layers.MaxPooling2D(pool_size=(strides[0], strides[0]))(x)
31.     x = layers.Dropout(0.1, name=prefix+'drop')((x))
32.     x = layers.BatchNormalization(name=prefix+'conv_bn')(x)
33.     return x
34.
35. def Transpose_Conv2D(x, filters, kernel, strides, padding, block_id, kernel_init=orthog
    onal()):
36.     prefix = f'block_{block_id}_'
37.     x = layers.Conv2DTranspose(filters, kernel_size=kernel, strides=strides, padding=pa
    dding,
38.                                kernel_initializer=kernel_init, name=prefix+'de-
    conv')(x)
39.     x = layers.LeakyReLU(name=prefix+'lrelu')(x)
40.     x = layers.Dropout(0.1, name=prefix+'drop')((x))
41.     x = layers.BatchNormalization(name=prefix+'conv_bn')(x)
42.     return x
43.
44.
45.
46. def AutoEncoder(input_shape):
47.     inputs = layers.Input(shape=input_shape)
48.
```

```
49.     # 240 x 240
50.     conv1 = Conv3DLayer(inputs, 16, (3,3,1), strides=(1,1,1), padding='same', block_id=
    1)
51.     conv2 = Conv3DLayer(conv1, 16, (3,3,1), strides=(2,2,1), padding='same', block_id=2
    )
52.
53.     # 120 x 120
54.     conv3 = Conv3DLayer(conv2, 32, (5,5,1), strides=(2,2,1), padding='same', block_id=3
    )
55.
56.     # 60 x 60
57.     conv4 = Conv3DLayer(conv3, 32, (3,3,1), strides=(1,1,1), padding='same', block_id=4
    )
58.     conv5 = Conv3DLayer(conv4, 64, (5,5,1), strides=(2,2,1), padding='same', block_id=5
    )
59.
60.     # 30 x 30
61.     conv6 = Conv3DLayer(conv5, 128, (3,3,1), strides=(2,2,1), padding='same', block_id=
    6)
62.     print("c1:",conv1.shape)
63.     print("c2:",conv2.shape)
64.     print("c3:",conv3.shape)
65.     print("c4:",conv4.shape)
66.     # 15 x 15
67.     meaner1=layers.Lambda(lambda x: K.mean(x, axis=3) )
68.     agglayer1 = meaner1(conv6)
69.     print(agglayer1.shape)
70.     deconv1 = Transpose_Conv2D(agglayer1, 128, 3, strides=2, padding='same', block_id=7
    )
71.
72.     # 30 x 30
73.     meaner2=layers.Lambda(lambda x: K.mean(x, axis=3) )
74.     agglayer2 = meaner2(conv5)
75.     print(agglayer2.shape)
76.     skip1 = layers.concatenate([deconv1, agglayer2], name='skip1')
77.     conv7 = Conv2DLayer(skip1, 64, 3, strides=(1,1,1), padding='same', block_id=8)
78.     deconv2 = Transpose_Conv2D(conv7, 32, 3, strides=2, padding='same', block_id=9)
79.
80.     # 60 x 60
81.     meaner3=layers.Lambda(lambda x: K.mean(x, axis=3) )
82.     agglayer3 = meaner3(conv3)
83.     print(agglayer3.shape)
84.     skip2 = layers.concatenate([deconv2, agglayer3], name='skip2')
85.     conv8 = Conv2DLayer(skip2, 32, 5, strides=(1,1,1), padding='same', block_id=10)
86.     deconv3 = Transpose_Conv2D(conv8, 16, 3, strides=2, padding='same', block_id=11)
87.
88.     # 120 x 120
89.     meaner4=layers.Lambda(lambda x: K.mean(x, axis=3) )
90.     agglayer4 = meaner4(conv2)
91.     print(agglayer4.shape)
92.     skip3 = layers.concatenate([deconv3, agglayer4], name='skip3')
93.     conv9 = Conv2DLayer(skip3, 16, 5, strides=(1,1,1), padding='same', block_id=12)
94.     deconv4 = Transpose_Conv2D(conv9, 16, 3, strides=2, padding='same', block_id=13)
95.
96.     # 240 x 240
97.     meaner5=layers.Lambda(lambda x: K.mean(x, axis=3) )
98.     agglayer5 = meaner5(conv1)
99.     print(agglayer5.shape)
100.        skip3 = layers.concatenate([deconv4, agglayer5])
101.        conv10 = layers.Conv2D(1, 3, strides=1, padding='same', activation='relu',
```

```
102.                                     kernel_initializer=orthogonal(), name='final_conv')(skip3
    )
103.
104.                return models.Model(inputs=inputs, outputs=conv10)
```

# Appendix B: Neural Network Model C Code

```
1.  #!/usr/bin/env python
2.  # coding: utf-8
3.  
4.  
5.  
6.  import tensorflow.keras
7.  from tensorflow.keras import backend as K
8.  import numpy as np
9.  import os
10. 
11. 
12. # # Model
13. 
14. 
15. import tensorflow.keras.layers as layers
16. import tensorflow.keras.models as models
17. from tensorflow.keras.initializers import orthogonal
18. 
19. 
20. def Conv3DLayer(x, filters, kernel, strides, padding, block_id, kernel_init=orthogonal(
    )):
21.     prefix = f'block_{block_id}_'
22.     x = layers.Conv3D(filters, kernel_size=kernel, strides=1, padding=padding,
23.                       kernel_initializer=kernel_init, name=prefix+'conv')(x)
24.     x = layers.LeakyReLU(name=prefix+'lrelu')(x)
25.     x = layers.MaxPooling3D(pool_size=(strides[0], strides[0], 1))(x)
26.     x = layers.Dropout(0.1, name=prefix+'drop')((x))
27.     x = layers.BatchNormalization(name=prefix+'conv_bn')(x)
28.     return x
29. 
30. def Conv2DLayer(x, filters, kernel, strides, padding, block_id, kernel_init=orthogonal(
    )):
31.     prefix = f'block_{block_id}_'
32.     x = layers.Conv2D(filters, kernel_size=kernel, strides=1, padding=padding,
33.                       kernel_initializer=kernel_init, name=prefix+'conv')(x)
34.     x = layers.LeakyReLU(name=prefix+'lrelu')(x)
35.     x = layers.MaxPooling2D(pool_size=(strides[0], strides[0]))(x)
36.     x = layers.Dropout(0.1, name=prefix+'drop')((x))
37.     x = layers.BatchNormalization(name=prefix+'conv_bn')(x)
38.     return x
39. 
40. def Transpose_Conv2D(x, filters, kernel, strides, padding, block_id, kernel_init=orthog
    onal()):
41.     prefix = f'block_{block_id}_'
42.     x = layers.Conv2DTranspose(filters, kernel_size=kernel, strides=strides, padding=pa
    dding,
43.                                kernel_initializer=kernel_init, name=prefix+'de-
    conv')(x)
44.     x = layers.LeakyReLU(name=prefix+'lrelu')(x)
45.     x = layers.Dropout(0.1, name=prefix+'drop')((x))
46.     x = layers.BatchNormalization(name=prefix+'conv_bn')(x)
47.     return x
48. 
49. 
50. 
51. def AutoEncoder(input_shape):
52.     inputs = layers.Input(shape=input_shape)
```

```
53.
54.     # 240 x 240
55.     conv1 = Conv3DLayer(inputs, 32, (3,3,1), strides=(1,1,1), padding='same', block_id=
    1)
56.     conv2 = Conv3DLayer(conv1, 32, (3,3,1), strides=(2,2,1), padding='same', block_id=2
    )
57.
58.     # 120 x 120
59.     conv3 = Conv3DLayer(conv2, 64, (5,5,1), strides=(2,2,1), padding='same', block_id=3
    )
60.
61.     # 60 x 60
62.     conv4 = Conv3DLayer(conv3, 64, (3,3,1), strides=(1,1,1), padding='same', block_id=4
    )
63.     conv5 = Conv3DLayer(conv4, 128, (5,5,1), strides=(2,2,1), padding='same', block_id=
    5)
64.
65.     # 30 x 30
66.     conv6 = Conv3DLayer(conv5, 256, (3,3,1), strides=(1,1,1), padding='same', block_id=
    6)
67.     print("c1:",conv1.shape)
68.     print("c2:",conv2.shape)
69.     print("c3:",conv3.shape)
70.     print("c4:",conv4.shape)
71.     # 30 x 30
72.     meaner1=layers.Lambda(lambda x: K.mean(x, axis=3) )
73.     agglayer1 = meaner1(conv6)
74.     print(agglayer1.shape)
75.     deconv1 = Transpose_Conv2D(agglayer1, 256, 3, strides=1, padding='same', block_id=7
    )
76.
77.     # 30 x 30
78.     meaner2=layers.Lambda(lambda x: K.mean(x, axis=3) )
79.     agglayer2 = meaner2(conv5)
80.     print(agglayer2.shape)
81.     skip1 = layers.concatenate([deconv1, agglayer2], name='skip1')
82.     conv7 = Conv2DLayer(skip1, 128, 3, strides=(1,1,1), padding='same', block_id=8)
83.     deconv2 = Transpose_Conv2D(conv7, 64, 3, strides=2, padding='same', block_id=9)
84.
85.     # 60 x 60
86.     meaner3=layers.Lambda(lambda x: K.mean(x, axis=3) )
87.     agglayer3 = meaner3(conv3)
88.     print(agglayer3.shape)
89.     skip2 = layers.concatenate([deconv2, agglayer3], name='skip2')
90.     conv8 = Conv2DLayer(skip2, 64, 5, strides=(1,1,1), padding='same', block_id=10)
91.     deconv3 = Transpose_Conv2D(conv8, 32, 3, strides=2, padding='same', block_id=11)
92.
93.     # 120 x 120
94.     meaner4=layers.Lambda(lambda x: K.mean(x, axis=3) )
95.     agglayer4 = meaner4(conv2)
96.     print(agglayer4.shape)
97.     skip3 = layers.concatenate([deconv3, agglayer4], name='skip3')
98.     conv9 = Conv2DLayer(skip3, 32, 5, strides=(1,1,1), padding='same', block_id=12)
99.     deconv4 = Transpose_Conv2D(conv9, 32, 3, strides=2, padding='same', block_id=13)
100.
101.            # 240 x 240
102.            meaner5=layers.Lambda(lambda x: K.mean(x, axis=3) )
103.            agglayer5 = meaner5(conv1)
104.            print(agglayer5.shape)
105.            skip3 = layers.concatenate([deconv4, agglayer5])
106.            conv10 = layers.Conv2D(1, 3, strides=1, padding='same', activation='relu',
```

```
107.                                 kernel_initializer=orthogonal(), name='final_conv')(skip3
    )
108. 
109.             return models.Model(inputs=inputs, outputs=conv10)
```

# Appendix C: Neural Network Training Code (Jupyter Notebook Format)

```
1.  Train Interpolation Neural Network
2.  import os
3.  os.environ['CUDA_DEVICE_ORDER'] = 'PCI_BUS_ID'
4.  os.environ['CUDA_VISIBLE_DEVICES'] = '0'
5.  os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
6.  import tensorflow.keras
7.  from tensorflow.keras import backend as K
8.  import numpy as np
9.  import matplotlib.pylab as plt
10. from tensorflow.keras.optimizers import SGD, Adamax
11. import Interp_Model_Long
12.
13. import Utilities
14. import SimpleITK as sitk  # For loading the dataset
15. import glob  # For populating the list of files
16. from scipy.ndimage import zoom  # For resizing
17. import re  # For parsing the filenames (to know their modality)
18. import random
19.
20. Load Data
21. # Get a list of files for all modalities individually
22. t1 = glob.glob('./original_data/*GG/*/*t1.nii.gz')
23. t2 = glob.glob('./original_data/*GG/*/*t2.nii.gz')
24. flair = glob.glob('./original_data/*GG/*/*flair.nii.gz')
25. t1ce = glob.glob('./original_data/*GG/*/*t1ce.nii.gz')
26. seg = glob.glob('./original_data/*GG/*/*seg.nii.gz')  # Ground Truth
27. pat = re.compile('.*_(\w*)\.nii\.gz')
28.
29. data_paths_orig = [{
30.     pat.findall(item)[0]:item
31.     for item in items
32. }
33. for items in list(zip(t1, t2, t1ce, flair, seg))]
34. np.size(data_paths_orig)
35. NumScans = 180
36. depthShrink = 7
37. wShrink= 0
38. hShrink= 0
39. inputData = np.zeros((NumScans*4*(77-depthShrink), 240-2*wShrink, 240-
    2*hShrink, 2), dtype='float32')
40. outputData = np.zeros((NumScans*4*(77-depthShrink), 240-2*wShrink, 240-
    2*hShrink), dtype='float32')
41. i = 0
42.
43. for j in range(NumScans):
44.     for k in (data_paths_orig[1]):
45.         if k not in 'seg':
46.             im_path1 = data_paths_orig[j][k]
47.             img1 = sitk.ReadImage(im_path1)
48.             arr1 = sitk.GetArrayFromImage(img1)
49.             if(arr1.shape[0] != 155):
50.                 print(arr1.shape)
51.                 print(im_path1)
52.                 continue
53.             outputData[i:i+77-depthShrink:1] = arr1[1+depthShrink:155-
depthShrink:2, wShrink:240-wShrink, hShrink:240-hShrink]
```

```
54.             inputData[i:i+77-depthShrink:1,:,:,0] = arr1[0+depthShrink:154-
    depthShrink:2, wShrink:240-wShrink, hShrink:240-hShrink]
55.             inputData[i:i+77-depthShrink:1,:,:,1] = arr1[2+depthShrink:155-
    depthShrink:2, wShrink:240-wShrink, hShrink:240-hShrink]
56.             i += 77-depthShrink
57.     print(j)
58. inputData = np.expand_dims(inputData, axis=-1)
59. outputData = np.expand_dims(outputData, axis=-1)
60. Randomize Data
61. random.Random(4).shuffle(inputData)
62. random.Random(4).shuffle(outputData)
63. Compile Model
64. model = Interp_Model_Long.AutoEncoder((240-2*wShrink,240-2*hShrink, 2, 1))
65. # model_opt = SGD(lr=0.005, decay=1-0.995, momentum=0.7, nesterov=False)
66. model_opt = Adamax(lr=0.0001)
67.
68. model.compile(optimizer=model_opt, loss='mse', metrics=['msle'])
69. %%capture cap --no-stderr
70. model.summary()
71. with open('output.txt', 'w') as f:
72.     f.write(cap.stdout)
73. Load Weights if restarting from interrupted training
74. model.load_weights('C:/Users/Gavin Karr/Programming/masters_research/dev/networks/scanN
    um180LongMod.hdf5')
75. Callbacks
76. modelchk = tensorflow.keras.callbacks.ModelCheckpoint('C:/Users/Gavin Karr/Programming/
    masters_research/dev/networks/scanNum180LongMod.hdf5',
77.                                       verbose=2,
78.                                       save_best_only=True,
79.                                       save_weights_only=False)
80.
81. tensorboard = tensorflow.keras.callbacks.TensorBoard(log_dir='logs',
82.                                           histogram_freq=0,
83.                                           write_graph=True,
84.                                           write_images=True)
85.
86. csv_logger = tensorflow.keras.callbacks.CSVLogger('./logs/keras_180LongMod.csv',
87.                                        append=True)
88. Train Model
89. hist = model.fit(x=inputData, y=outputData, batch_size=4, epochs=300, validation_split=
    0.2, callbacks=[modelchk, tensorboard, csv_logger])
90. # save learning curves data
91. import pandas as pd
92. learning_curves = pd.DataFrame()
93. learning_curves['epoch'] = hist.epoch
94. learning_curves['loss'] = hist.history['loss']
95. learning_curves['val_loss'] = hist.history['val_loss']
96.
97. #learning_curves.to_csv('./RemovingNoiseFromMusic.csv')
98. Utilities.PlotLearningCurves(learning_curves)
99. Quick Test Results
100.       tesOut = np.zeros((240-2*wShrink, 240-2*hShrink), dtype='uint16')
101.       tesIn1 = np.zeros((240-2*wShrink, 240-2*hShrink, 2), dtype='uint16')
102.       sliceNum = 25
103.       patNum = 162
104.       modal = 't1'
105.
106.       im_path1 = data_paths_orig[patNum][modal]
107.       img1 = sitk.ReadImage(im_path1)
108.       arr1 = sitk.GetArrayFromImage(img1)
109.
```

```
110.        tesOut = arr1[sliceNum, wShrink:240-wShrink, hShrink:240-hShrink]
111.        tesIn1[:,:,0] = arr1[sliceNum-3, wShrink:240-wShrink, hShrink:240-hShrink]
112.        tesIn1[:,:,1] = arr1[sliceNum+3, wShrink:240-wShrink, hShrink:240-hShrink]
113.        outD = np.expand_dims(tesIn1, axis=0)
114.        outD = np.expand_dims(outD, axis=-1)
115.        gtD = tesOut
116.
117.        in1 = outD[0,:,:,0,0]
118.        in2 = outD[0,:,:,1,0]
119.
120.        outT= model.predict(outD)
121.
122.        import scipy.ndimage as ndimage
123.
124.        # given 2 arrays arr1, arr2
125.        arr1 = tesIn1[:,:,0]
126.        arr2 = tesIn1[:,:,1]
127.
128.        # rejoin arr1, arr2 into a single array of shape (2, 10, 10)
129.        arrn = np.r_['0,3', arr1, arr2]
130.
131.        # define the grid coordinates where you want to interpolate
132.        X, Y = np.meshgrid(np.arange(240-2*wShrink), np.arange(240-2*wShrink))
133.        # 0.5 corresponds to half way between arr1 and arr2
134.        coordinates = np.ones((240-2*wShrink,240-2*hShrink))*0.5, X, Y
135.
136.        # given arr interpolate at coordinates
137.        newarr = ndimage.map_coordinates(arrn, coordinates, order=2).T
138.
139.
140.        finIm = outT[0,:,:,0]
141.        # Plot new array
142.        fig, ax = plt.subplots(ncols=5)
143.        cmap = plt.get_cmap('Greys')
144.
145.        vmin = np.min([in1.min(), finIm.min(), in2.min()])
146.        vmax = np.max([in1.max(), finIm.max(), in2.max()])
147.        ax[0].imshow(in1, interpolation='nearest', cmap=cmap, vmin=vmin, vmax=vmax)
148.        ax[1].imshow(newarr, interpolation='nearest', cmap=cmap, vmin=vmin, vmax=vmax)
149.        ax[2].imshow(finIm, interpolation='nearest', cmap=cmap, vmin=vmin, vmax=vmax)
150.        ax[3].imshow(gtD, interpolation='nearest', cmap=cmap, vmin=vmin, vmax=vmax)
151.        ax[4].imshow(in2, interpolation='nearest', cmap=cmap, vmin=vmin, vmax=vmax)
152.        ax[0].set_xlabel('Bottom Slice')
153.        ax[1].set_xlabel('Interpolated')
154.        ax[2].set_xlabel('Predicted')
155.        ax[3].set_xlabel('Ground Truth')
156.        ax[4].set_xlabel('Top Slice')
157.        plt.show()
158.        print("linear interp is:", Utilities.MSE(newarr, gtD))
159.        print("CNN predicted is:", Utilities.MSE(finIm, gtD))
160.        imNum = 1579
161.        outD = np.expand_dims(inputData[imNum], axis=0)
162.        gtD = outputData[imNum,:,:,0]
163.
164.        in1 = outD[0,:,:,0,0]
165.        in2 = outD[0,:,:,1,0]
166.
167.        outT= model.predict(outD)
168.
169.        finIm = outT[0,:,:,0]
170.        # Plot new array
```

```
171.        fig, ax = plt.subplots(ncols=4)
172.        cmap = plt.get_cmap('Greys')
173.
174.        vmin = np.min([in1.min(), finIm.min(), in2.min()])
175.        vmax = np.max([in1.max(), finIm.max(), in2.max()])
176.        ax[0].imshow(in1, interpolation='nearest', cmap=cmap, vmin=vmin, vmax=vmax)
177.        ax[1].imshow(finIm, interpolation='nearest', cmap=cmap, vmin=vmin, vmax=vmax)
178.        ax[2].imshow(gtD, interpolation='nearest', cmap=cmap, vmin=vmin, vmax=vmax)
179.        ax[3].imshow(in2, interpolation='nearest', cmap=cmap, vmin=vmin, vmax=vmax)
180.        ax[0].set_xlabel('arr1')
181.        ax[1].set_xlabel('interpolated')
182.        ax[2].set_xlabel('Ground Truth')
183.        ax[3].set_xlabel('arr2')
184.        plt.show()
185.        Generate Model Summary
186.        model.summary()
187.        %%capture cap --no-stderr
188.        print(model.summary())
189.        Generate Model Graph
190.        from tensorflow.keras.utils import plot_model
191.        plot_model(model, to_file='model.png')
```

## Appendix D: Test Neural Network Performance (Jupyter Notebook Format)

```
1.  Test_CNN
2.  Test Neural Net Performance¶
3.  In [ ]:
4.  import os
5.  os.environ['CUDA_DEVICE_ORDER'] = 'PCI_BUS_ID'
6.  os.environ['CUDA_VISIBLE_DEVICES'] = '0'
7.  os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
8.  import tensorflow.keras
9.  from tensorflow.keras import backend as K
10. import numpy as np
11. import matplotlib.pylab as plt
12. from tensorflow.keras.optimizers import SGD, Adamax
13. import Interp_Model_WBN_L
14. import Interp_Model
15. import Interp_Model_Long
16.
17. import Utilities
18. import SimpleITK as sitk  # For loading the dataset
19. import glob  # For populating the list of files
20. from scipy.ndimage import zoom  # For resizing
21. import re  # For parsing the filenames (to know their modality)
22. import random
23. In [ ]:
24. import numpy as np
25. def calculate_psnr(img1, img2, max_value=255):
26.     """"Calculating peak signal-to-noise ratio (PSNR) between two images."""
27.     mse = np.mean((np.array(img1, dtype=np.float32) - np.array(img2, dtype=np.float32))
     ** 2)
28.     if mse == 0:
29.         return 1000
30.     return 20 * np.log10(max_value / (np.sqrt(mse)))
31. Load Data¶
32. In [ ]:
33. # Get a list of files for all modalities individually
34. t1 = glob.glob('./original_data/*GG/*/*t1.nii.gz')
35. t2 = glob.glob('./original_data/*GG/*/*t2.nii.gz')
36. flair = glob.glob('./original_data/*GG/*/*flair.nii.gz')
37. t1ce = glob.glob('./original_data/*GG/*/*t1ce.nii.gz')
38. seg = glob.glob('./original_data/*GG/*/*seg.nii.gz')  # Ground Truth
39. In [ ]:
40. pat = re.compile('.*_(\w*)\.nii\.gz')
41.
42. data_paths_orig = [{
43.     pat.findall(item)[0]:item
44.     for item in items
45. }
46. for items in list(zip(t1, t2, t1ce, flair, seg))]
47. In [ ]:
48. NumScans = 5
49. depthShrink = 0
50. wShrink= 0
51. hShrink= 0
52. In [ ]:
53. inputData = np.zeros((NumScans*4*(77-depthShrink), 240-2*wShrink, 240-
    2*hShrink, 2), dtype='uint16')
54. outputData = np.zeros((NumScans*4*(77-depthShrink), 240-2*wShrink, 240-
    2*hShrink), dtype='uint16')
```

```
55. In [ ]:
56. i = 0
57.
58. for j in range(180,185):
59.     for k in (data_paths_orig[1]):
60.         if k not in 'seg':
61.             im_path1 = data_paths_orig[j][k]
62.             img1 = sitk.ReadImage(im_path1)
63.             arr1 = sitk.GetArrayFromImage(img1)
64.             if(arr1.shape[0] != 155):
65.                 print(arr1.shape)
66.                 print(im_path1)
67.                 continue
68.             outputData[i:i+77-depthShrink:1] = arr1[1+depthShrink:155-
    depthShrink:2, wShrink:240-wShrink, hShrink:240-hShrink]
69.             inputData[i:i+77-depthShrink:1,:,:,0] = arr1[0+depthShrink:154-
    depthShrink:2, wShrink:240-wShrink, hShrink:240-hShrink]
70.             inputData[i:i+77-depthShrink:1,:,:,1] = arr1[2+depthShrink:155-
    depthShrink:2, wShrink:240-wShrink, hShrink:240-hShrink]
71.             i += 77-depthShrink
72.     print(j)
73. In [ ]:
74. inputData = np.expand_dims(inputData, axis=-1)
75. outputData = np.expand_dims(outputData, axis=-1)
76. Load and Run Model¶
77. In [ ]:
78. model = Interp_Model_Long.AutoEncoder((240-2*wShrink,240-2*hShrink, 2, 1))
79. In [ ]:
80. model.load_weights('./networks/scanNum180LongMod.hdf5')
81. In [ ]:
82. outT= model.predict(inputData, batch_size=20)
83. Perform Linear Interpolation and Calculate MSE¶
84. In [ ]:
85. lin_interpMSE = 0
86. CNN_interpMSE = 0
87.
88. for i in range(inputData.shape[0]):
89.    # im_path1 = data_paths_orig[patNum][modal]
90.     #img1 = sitk.ReadImage(im_path1)
91.    # imArray = sitk.GetArrayFromImage(img1)
92.     #for j in range(1, 154):
93.     tesOut = np.zeros((240-2*wShrink, 240-2*hShrink), dtype='uint16')
94.     tesIn1 = np.zeros((240-2*wShrink, 240-2*hShrink, 2), dtype='uint16')
95.     #sliceNum = j
96.     #patNum = i
97.    # modal = 't1'
98.
99.
100.            tesOut = outputData[i, wShrink:240-wShrink, hShrink:240-hShrink, 0]
101.            tesIn1[:,:,0] = inputData[i, wShrink:240-wShrink, hShrink:240-
    hShrink, 0, 0]
102.            tesIn1[:,:,1] = inputData[i, wShrink:240-wShrink, hShrink:240-
    hShrink, 1, 0]
103.
104.            outD = np.expand_dims(tesIn1, axis=0)
105.            outD = np.expand_dims(outD, axis=-1)
106.            gtD = tesOut
107.
108.            in1 = outD[0,:,:,0,0]
109.            in2 = outD[0,:,:,1,0]
110.
```

```
111.
112.            import scipy.ndimage as ndimage
113.
114.            # given 2 arrays arr1, arr2
115.            arr1 = tesIn1[:,:,0]
116.            arr2 = tesIn1[:,:,1]
117.
118.            # rejoin arr1, arr2 into a single array of shape (2, 10, 10)
119.            arrn = np.r_['0,3', arr1, arr2]
120.
121.            # define the grid coordinates where you want to interpolate
122.            X, Y = np.meshgrid(np.arange(240-2*wShrink), np.arange(240-2*wShrink))
123.            # 0.5 corresponds to half way between arr1 and arr2
124.            coordinates = np.ones((240-2*wShrink,240-2*hShrink))*0.5, X, Y
125.
126.            # given arr interpolate at coordinates
127.            newarr = ndimage.map_coordinates(arrn, coordinates, order=2).T
128.            finIm = outT[i,:,:,0]
129.
130.            lin_interpMSE += Utilities.MSE(newarr, gtD)
131.            CNN_interpMSE += Utilities.MSE(finIm, gtD)
132.            if(i % 1000 == 0):
133.                print(i)
134.        In [ ]:
135.        print("linear interp is:", lin_interpMSE)
136.        print("CNN predicted is:", CNN_interpMSE)
137.        Graphically Display Results¶
138.        In [ ]:
139.        i=860
140.        tesIn1 = np.zeros((240-2*wShrink, 240-2*hShrink, 2), dtype='uint16')
141.        tesIn1[:,:,0] = inputData[i, wShrink:240-wShrink, hShrink:240-hShrink, 0, 0]
142.        tesIn1[:,:,1] = inputData[i, wShrink:240-wShrink, hShrink:240-hShrink, 1, 0]
143.
144.        outD = np.expand_dims(tesIn1, axis=0)
145.        outD = np.expand_dims(outD, axis=-1)
146.        gtD = tesOut
147.
148.        in1 = outD[0,:,:,0,0]
149.        in2 = outD[0,:,:,1,0]
150.
151.
152.        import scipy.ndimage as ndimage
153.
154.        # given 2 arrays arr1, arr2
155.        arr1 = tesIn1[:,:,0]
156.        arr2 = tesIn1[:,:,1]
157.
158.        # rejoin arr1, arr2 into a single array of shape (2, 10, 10)
159.        arrn = np.r_['0,3', arr1, arr2]
160.
161.        # define the grid coordinates where you want to interpolate
162.        X, Y = np.meshgrid(np.arange(240-2*wShrink), np.arange(240-2*wShrink))
163.        # 0.5 corresponds to half way between arr1 and arr2
164.        coordinates = np.ones((240-2*wShrink,240-2*hShrink))*0.5, X, Y
165.        newarr = ndimage.map_coordinates(arrn, coordinates, order=2).T
166.
167.        fig, ax = plt.subplots(ncols=5, figsize=(18, 20))
168.        cmap = plt.get_cmap('Greys')
169.        for a in ax:
170.            a.set_xticks([])
171.            a.set_yticks([])
```

```
172.
173.
174.        vmin = np.min([in1.min(), finIm.min(), in2.min()])
175.        vmax = np.max([in1.max(), finIm.max(), in2.max()])
176.        ax[0].imshow(inputData[i, wShrink:240-wShrink, hShrink:240-
    hShrink, 0, 0], interpolation='nearest', cmap=cmap, vmin=vmin, vmax=vmax)
177.        ax[1].imshow(newarr, interpolation='nearest', cmap=cmap, vmin=vmin, vmax=vmax)
178.        ax[2].imshow(outT[i,:,:,0], interpolation='nearest', cmap=cmap, vmin=vmin, vmax=
    vmax)
179.        ax[3].imshow(outputData[i, wShrink:240-wShrink, hShrink:240-
    hShrink, 0], interpolation='nearest', cmap=cmap, vmin=vmin, vmax=vmax)
180.        #ax[3].imshow(newarr - outT[i,:,:,0], interpolation='nearest', cmap=cmap, vmin=v
    min, vmax=vmax)
181.        ax[4].imshow(inputData[i, wShrink:240-wShrink, hShrink:240-
    hShrink, 1, 0], interpolation='nearest', cmap=cmap, vmin=vmin, vmax=vmax)
182.        ax[0].set_xlabel('Bottom Slice')
183.        ax[1].set_xlabel('Interpolated')
184.        ax[2].set_xlabel('Predicted')
185.        ax[3].set_xlabel('Ground Truth')
186.        ax[4].set_xlabel('Top Slice')
187.        plt.show()
188.        Calculate PSNR¶
189.        In [ ]:
190.        lin_interpPSNR = 0
191.        CNN_interpPSNR = 0
192.
193.        for i in range(inputData.shape[0]):
194.           # im_path1 = data_paths_orig[patNum][modal]
195.            #img1 = sitk.ReadImage(im_path1)
196.           # imArray = sitk.GetArrayFromImage(img1)
197.            #for j in range(1, 154):
198.            tesOut = np.zeros((240-2*wShrink, 240-2*hShrink), dtype='uint16')
199.            tesIn1 = np.zeros((240-2*wShrink, 240-2*hShrink, 2), dtype='uint16')
200.            #sliceNum = j
201.            #patNum = i
202.           # modal = 't1'
203.
204.
205.            tesOut = outputData[i, wShrink:240-wShrink, hShrink:240-hShrink, 0]
206.            tesIn1[:,:,0] = inputData[i, wShrink:240-wShrink, hShrink:240-
    hShrink, 0, 0]
207.            tesIn1[:,:,1] = inputData[i, wShrink:240-wShrink, hShrink:240-
    hShrink, 1, 0]
208.
209.            outD = np.expand_dims(tesIn1, axis=0)
210.            outD = np.expand_dims(outD, axis=-1)
211.            gtD = tesOut
212.
213.            in1 = outD[0,:,:,0,0]
214.            in2 = outD[0,:,:,1,0]
215.
216.
217.            import scipy.ndimage as ndimage
218.
219.            # given 2 arrays arr1, arr2
220.            arr1 = tesIn1[:,:,0]
221.            arr2 = tesIn1[:,:,1]
222.
223.            # rejoin arr1, arr2 into a single array of shape (2, 10, 10)
224.            arrn = np.r_['0,3', arr1, arr2]
225.
```

```
226.            # define the grid coordinates where you want to interpolate
227.            X, Y = np.meshgrid(np.arange(240-2*wShrink), np.arange(240-2*wShrink))
228.            # 0.5 corresponds to half way between arr1 and arr2
229.            coordinates = np.ones((240-2*wShrink,240-2*hShrink))*0.5, X, Y
230.
231.            # given arr interpolate at coordinates
232.            newarr = ndimage.map_coordinates(arrn, coordinates, order=2).T
233.            finIm = outT[i,:,:,0]
234.            ps1 = calculate_psnr(gtD, newarr, max_value=65535)
235.            ps2 = calculate_psnr(gtD, finIm, max_value=65535)
236.
237.            if(ps1 < 1000 and ps1 > -100):
238.                lin_interpPSNR+= ps1
239.            if(ps2 < 1000 and ps2 > -100):
240.                CNN_interpPSNR+= ps2
241.            if(i % 1000 == 0):
242.                print(i)
243.        In [ ]:
244.        print("linear interp PSNR is:", lin_interpPSNR/inputData.shape[0])
245.        print("CNN predicted PSNR is:", CNN_interpPSNR/inputData.shape[0])
246.        In [ ]:
247.
```