Master Thesis 2020 ,30 ECTS
Faculty of Science and Techonology
Professor Cecilia Marie Futsæther

Automated volumetric delineation of cancer tumors on PET/CT images using 3D convolutional neural network (V-Net)

Afreen Mirza
MSc Data Science

This page is intentionally left blank.

Acknowledgments

Foremost, it gives me great pleasure in expressing my heartfelt gratitude to my advisor, Prof. Cecilia Marie Futsæther, for her immense support, valuable feedback, and help during this project.

Furthermore, I want to thank the scientists who participated in all the meetings regarding my project and came with helpful feedback: Ms. Aurora Grøndahl and Prof. Oliver Tomic.

I want to thank Mr. Yngve Mardal Moe for helping me throughout my thesis for programmatical and technological guidance, which helped me in implementing the project successfully.

Next, I am incredibly thankful to all my friends who helped me in this thesis and, in particular, Ms. Bao Ngoc Huynh for her support at various stages of the project.

I thank my husband, Imroj Sheikh, who supported me through long days of writing and helping throughout this period in completing the thesis.

I want to thank my parents for encouraging and supporting me through this thesis and life in general.

Afreen Mirza
Ås, 2nd July, 2020

Abstract

Purpose
The process of delineation of tumors and malignant lymph nodes using medical images is a fundamental part of radiotherapy planning. Still, this process is done manually by radiologists. This process is time-consuming and suffers from inter-observer variability. Hence, there is a need to fully automate this process of delineation to reduce time consumption and inter-observer variability. Deep learning is a division of artificial intelligence that has proven to be useful for the automatic segmentation of medical images with the use of neural networks. We have to follow a systematic procedure as these neural networks require a large number of parameters to be tuned during the delineation process, to guarantee reproducibility. This thesis will present a complete theory of deep learning and a convolutional neural network, for delineating 3D images. The project will use the *deoxys* framework to implement the V-Net architecture for automatic delineation of gross tumor volume and malignant lymph nodes in the head and neck region.

Methods
The project uses the *deoxys* framework developed by *Ngoc Huynh Bao* for implementing a V-Net architecture for automatic delineation of 3D images to segment cancer tumors and lymph nodes. This implementation includes designing a deep learning model that is parametrized by a series of JSON configuration files that contain the model hyperparameters. It also includes converting the dataset files into the HDF5 file format which is done using an HDF5 Data-reader, which is an accessible file format for storing massive data.

The dataset consists of medical images of 197 patients who have undergone treatment at Oslo University Hospital, The Radium Hospital. The images are a combination of contrast-enhanced CT scans and PET co-registered (i.e., contrast-enhanced PET/CT scans) that are available for all patients. The dataset is divided into a training set (142 patients), a validation set (15 patients), and a test set (40 patients), without any stratification of tumor stages.

All the models used in the project are based on the V-Net architecture used for 3D images. Dice loss, as well as binary F_β loss, are introduced in the experiments. For optimizing the loss, Adam optimizer is introduced. Models are tested only for the standard convolutional layers. All the models were compared based on the average Dice per image in the validation set. Only the highest performing models using the PET/CT information were used to delineate the test set.

Results
The 3D convolutions have a higher memory footprint due to the use of 3D image volumes. Hence, all the experiments were performed on the Orion cluster which is an open-source platform hosted by NMBU and operated by CIGENE.

Orion is a remote server that helps a user to run experiments with large CPU memory and GPU's for fast processing speed and to meet the memory issues with the use of a single GPU. The experiments were run with different parameter combinations and with different set of filters in the downsampling and upsampling layers of the model. The model that we used consisted of [32, 64, 128, 256, 512] filters in the V-Net architecture. The F_β loss was better than the Dice loss and gave a good overall performance on the validation set. The highest performing PET/CT model gave F_β score of 0.6750 and Dice score of 0.6286 on the test set. Tensorboard logger is used for automatic performance logging.

Conclusion

In this thesis, we implemented a V-Net model using the *deoxys* framework for tumor segmentation of 3D PET/CT images of head and neck cancer patients. This project makes use of 3D convolutions operations to take complete advantage of volumetric information for multi-modality images. We successfully created an HDF5 data-Reader for handling massive image data.

Previous studies have shown that deep learning can be very consistent, time-saving in the medical image analysis field, and for the segmentation of tumors and malignant lymph nodes tissue in HNC patients. The 3D V-net model has shown an adequate performance and can be a preferable choice over the 2D convolution networks. However, our proposed model does not reach the expected Dice performance, so we cannot conclude that the automatically generated segmentation maps are similar to those produced by radiologists. Still, deep learning has a vast potential, which can considerably change the way of delineation presently done by radiologists and can serve as a second opinion in the delineation process.

Contents

List of Figures

List of Tables

Abbreviations

Abbreviation	Meaning
CT	Computed tomography
PET	Positron emission tomography
MRI	Magnetic resonance imaging
CPU	Central proccessing unit
GPU	Graphics processing unit
OAR	Organs at risk
HNC	Head and neck cancer
CTV	Clinical target volume
GTV	Gross tumor volume
ML	Machine Learning
MLP	Multilayer perceptron
DLC	Deep Learning classifier
CNN	Convolutional neural network
ReLU	Rectified Linear Units
SGD	Stochastic Gradient Descent
BN	Batch normalization
TP and FP	True positive and False positive
TN and FN	True negative and False negative
TPR	True positive rate
TNR	True negative rate
PPV	Positive predictive value
DSC	Dice similarity coefficient
HDF5	Hierarchical data format
NMBU	Norwegian University of Life Sciences

Chapter 1

Introduction

1.1 Motivation

Worldwide, cancer is the second leading cause of deaths, accounting for about 9.6 million deaths in 2018 [1]. There is a rise in cancer in many parts of the world due to lack of early detection tools and screening programs that will help to detect the disease early. Also, the high price associated with any cancer detection program that helped many patients in wealthier countries is not readily available in other parts of the world [2]. There is a need for creating different detection techniques [2] that are less expensive and time-consuming which will help in curing cancer at an early stage.

The most popular treatment for cancer, along with chemotherapy and surgery, is radiotherapy, which is very cost-effective [3]. Radiotherapy is given to cancer patients in the form of ionizing radiation that kills the cancer cells. Still, this radiation dose can damage the healthy tissues around the cancer cells. One main aim of health professionals in any cancer treatment planning is to identify the boundary of cancer tumors such that the maximum radiation dose is given to abnormal cells and hence minimize damage to healthy tissues [3]. Various image-guided modalities are used for understanding the structural information [4], the exact location of cancer tumors inside the patient body. Computed Tomography (CT Scan), Positron Emission Tomography (PET scan), Magnetic Resonance Imaging (MRI), and ultrasound are among the widely used imaging techniques in the medical imaging field.

1.1.1 Challenges with cancer treatment

The medical imaging field generates a tremendous amount of image data, which is segmented manually by an expert to delineate the boundary of the cancer tumor. The total number of medical images generated by the radiology department in every hospital is so vast, up to 3000 images [5], [6], which can be around 250 GB of data [6], to analyze. This manual segmentation requires a considerable amount of time.

In addition, the amount of data generated in the form of images is so vast that processing and scaling them at a higher speed is not possible by a radiologist [6]. Therefore, there is a need for automation of this processing using some computer-aided program such as Deep Learning [7], [8]. Deep learning can help perform this processing more efficiently and in comparatively less time.

In medical image analysis, time plays a significant factor in the diagnosis and early detection of cancer, which can save a patient's life and provide them a longer life. The new advancement in the field of deep learning and its frameworks has made this processing more efficient, accurate, and fast. Also, the advanced, powerful CPU (central processing unit) and GPU (graphics processing unit) processing power is very helpful for radiologists for scaling their diagnostic results.

1.1.2 Methods used for automated delineation of head and neck cancer tumor

Head and neck cancers are the type of cancer that originates in the larynx (voice box), throat, lips, mouth, nose, and salivary glands [9]. These types of cancers also contain malignant lymph nodes [10], which also have to be delineated by the radiologists.

Radiologists delineate the boundary of the cancer tumors and specific organs at risk (OAR), such that OAR receives a small dosage of radiation and can be protected from the ill effects of radiotherapy [11]. There are several imaging modalities available to delineate tumors such as CT, MRI, and PET in HNC patients. Accurate delineation of the tumor volume is a crucial task in HNC patients. If the target volume is not appropriately delineated, it can cause damage to healthy tissues [11]. Also, delineation process suffers from inter-intraobserver variability (Multi-Institutional Target Delineation in Oncology, Hong *et al.*) [10], which arises when different radiologists delineate different tumor boundaries.

There have been several efforts made to automate this process , which are being developed based on handcrafted approaches as well as machine learning approaches. Semi-automatic graph-based algorithms such as graph-cut [13] and Markov random field [14] , [15] implemented especially on PET images. In 2009 Yu *et al.* [16] developed a machine learning decision tree-based algorithm based on local texture features to segment HNC tumors. However, the performance accuracy of these models is minimal.

To fully automate this delineation process, different deep learning approaches have proven to be very beneficial. In this project, we use algorithms based on a convolutional neural network that have demonstrated great success in different computer vision tasks. For example, consider two deep learning studies done by Cardenas *et al.*, used for the delineation of malignant tissue for HNC patients [10], [17]. Both studies used manually segmented clinical target volume (CTV)and gross tumor volume (GTV) of CT images. They both used a different model one uses stacked two-layer auto-encoder [17], and the other uses 3D CNN [10]. But the performance of both has shown a high Dice score ranging between 0.70 - 0.85. Many other studies have demonstrated good performance of deep learning approaches in delineating cancer tumors such as nodules in the lungs [18], [19]. Deep learning approaches can be very beneficial for segmenting tumor volumes and organs at risk and reduces the segmentation time by automating the process.

1.1.3 Problem statement

This project aims to provide an understanding of automation methods available for the delineation of cancer tumors using deep learning techniques to the reader.

The first goal of the thesis is to introduce different concepts of deep learning for the segmentation of images to the reader. Readers should have a basic understanding of standard 'machine learning' concepts and linear algebra (for understanding vectors, matrices, matrix multiplication), also some knowledge of calculus (for understanding differentiation used in some sections).

The second objective is to develop a deep learning framework using image segmentation algorithms for automating the delineation process of cancer tumors. The design of the project is chosen in such a way that it is reproducible and created using a standardized method for performing different experiments. Also, there is an automatic performance logging possible for different parameters and results.

The final task is to test the developed framework for the automatic segmentation of tumors and malignant lymph nodes. The input to the network used for delineation will be 3D PET/CT images of head and neck cancer. At last, the benefit of the 3D model used for segmentation is compared with the 2D delineation framework performance.

Chapter 2

Deep learning

2.1 Deep Learning

2.1.1 Introduction to Deep Learning

Deep learning is an artificial intelligence technique and a branch of machine learning [20], [21] which has gained popularity in various computer vision tasks such as object detection, image classification, and semantic segmentation.

Different models consisting of several preprocessing layers [21], can quickly learn patterns in data with the help of deep learning models, which is very useful in identifying and quantifying different patterns in medical images.

Medical image segmentation works well with machine learning for segmenting the healthier region as compared to the diseased area [20]. But a typical machine-learning segmentation base model requires preprocessing steps (removing noise, contrast enhancement), feature extraction techniques, and these extracted features are fed to a machine-learning model.

On the contrary, a deep learning model does not require any preprocessing, segmentation, or feature extraction [20]. Images can be processed directly. Due to limitations on the size of input images, sometimes they are resized before being fed to the model. DLC (Deep learning classifier) has the capability of avoiding any errors resulting from feature vectors or imprecise segmentation, and hence they have an excellent classification accuracy. Figure 2.1 represents the comparison between

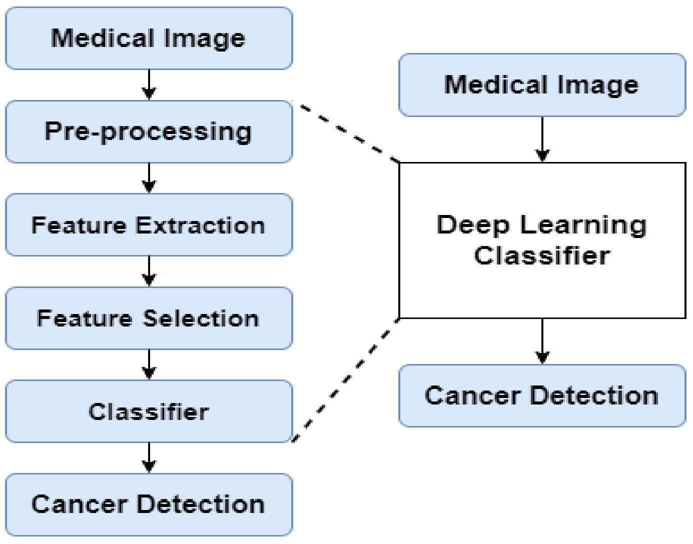


Figure 2.1: Different classifier approach using typical ML algorithm and DLC [20].

the two different approaches used for image segmentation ML (Machine learning) and DLC. The presence of several hidden layers inside DLC networks makes them computationally very intensive [20]. DLC works on hierarchical feature learning and higher feature extraction and abstraction level, which has contributed to their success in the field of artificial intelligence.

2.1.2 Artificial neural networks

This section will build an intuition on the components required for constructing a neural network for semantic image segmentation.

2.1.3 Multilayer perceptron

A multilayer perceptron as shown in Figure 2.2 is a fully connected neural network where each input unit connects to every node in succeeding layers [22]. It consists of an input layer to receive the different input signals and an output layer that predicts outputs based on the input provided, and in between these two layers, there are one or more hidden layers, that are the actual computational engines of the MLP (Multilayer perceptron). If such a network consists of more than one hidden layer, it is known as a *deep artificial neural network* [8].

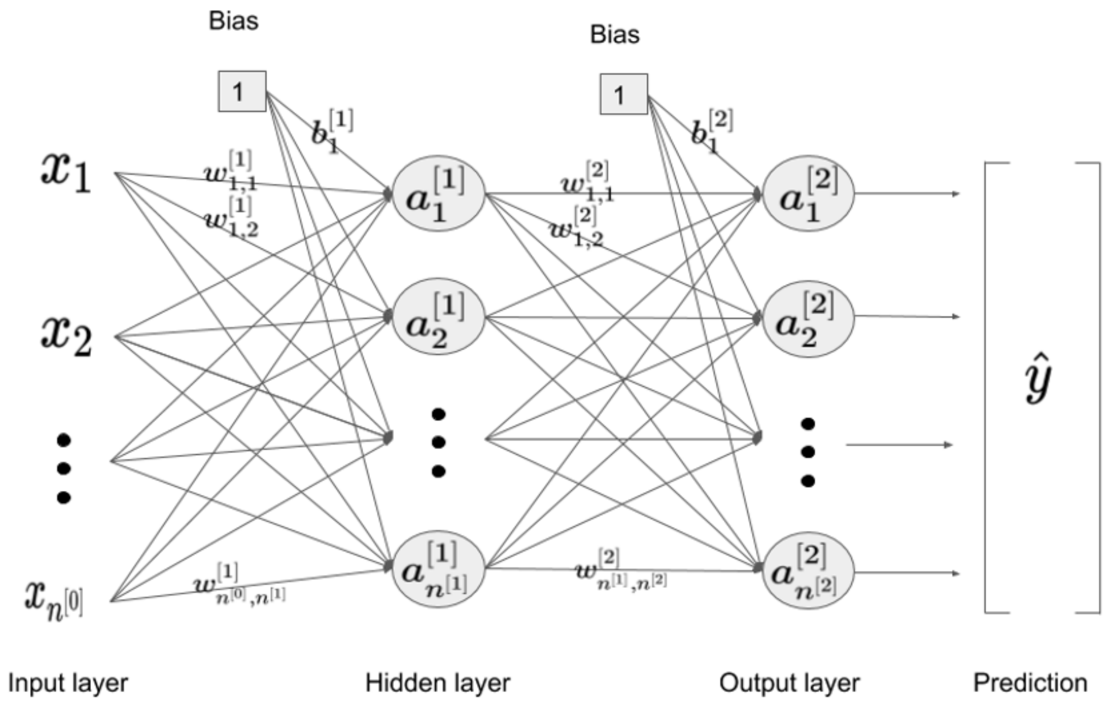


Figure 2.2: Illustration of a multilayer perceptron where a represents an activation unit, w is the weight of a connection, n is the number of units in the layer, and b is the bias. The input layer represents layer 'zero', the hidden layer is layer 'one', and the output layer is layer 'two'. $a_2^{[1]}$ indicates the activation of unit two in layer one. Similarly, $w_{1,1}^{[1]}$ describes the weight of the connection between layer zero and node one in layer one [8].

Each perceptron unit in a given layer receives input from all the units in the preceding layer. A net-input is calculated by matrix multiplication of input values with weights, added with a bias. We compute the final output of the activation unit by applying a non-linear differentiable activation function as described in section 2.1.6, that introduces non-linearity to the MLP. This function should be differentiable to update the weights during backpropagation as discussed in section 2.1.19 .

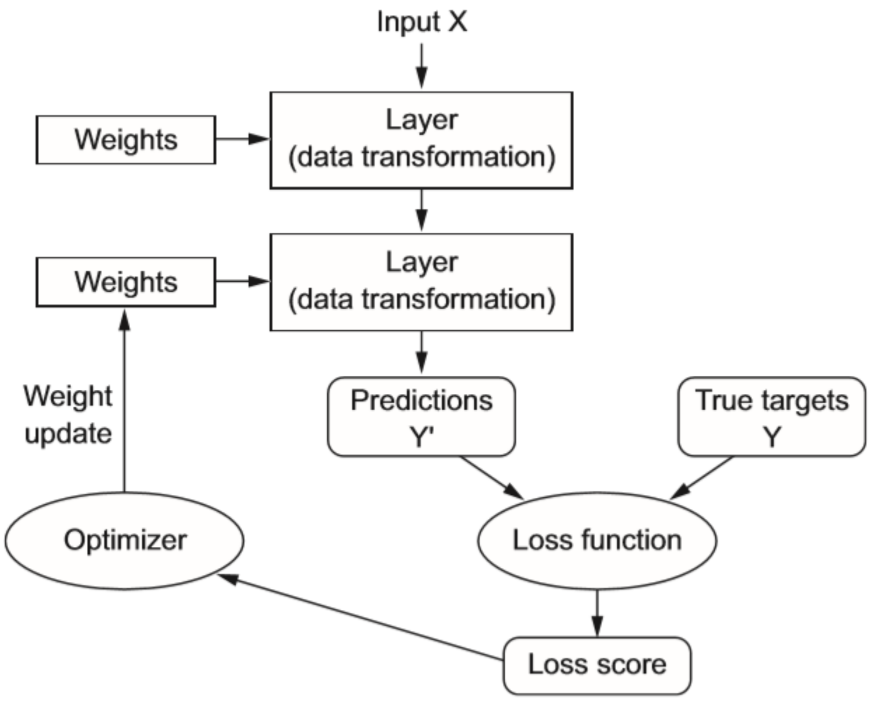


Figure 2.3: Illustration of relationship between layers, optimizer and loss function of neural network [7].

2.1.4 Forward Propagation: Activating a neural network

The forward propagation starts at the input layer, where the patterns of training input data X are propagated through the network to generate predicted output $\widehat{Y}$ as shown in Figure 2.3. Based on the output of the network, the loss-functions as discussed in section 2.1.5 compares the prediction to the true targets generating a loss value or error which we want to minimize. This loss value signifies how well the model prediction matches the actual target Y [7]. At last, the optimizer uses this loss value to update the network's weight.

By referring Figure 2.2, the net-input $Z^{(h)}$ and activation $A^{(h)}$ of the hidden layer h by arranging the weights, activations and bias (b^h) [8] for an input containing n training samples, can be expressed as :

$$Z^{(h)} = W^{(h)} A^{(input)} + b^h \tag{2.1}$$

$A^{(input)}$ is a (n x p) matrix, and the matrix-matrix multiplication with weights results in an (n x k) dimensional net-input matrix $Z^{(h)}$. Lastly, activation function $\phi(.)$ is applied to each value in the net-input matrix to get the n x k activation matrix $A^{(h)}$ for the next layer (here, the output layer) [8]:

$$A^{(h)} = \phi\left(Z^{(h)}\right) \tag{2.2}$$

Here, h subscript denotes hidden layer. Activation function $\phi(.)$ can be chosen from all the available functions for neural-networks discussed in section 2.1.6.

2.1.5 Loss Functions

The loss function, also known as the objective function, plays an essential role in any deep neural network. The loss function observes the predictions obtained from the model output value and the true target value (the value the user wants to achieve from the network) [23]. A score is calculated between them known as the distance score, which signifies how well the network has performed for a given problem. Any deep learning model aims to yield weight values that decrease the loss during a repeated loop of training. The type of loss function chosen can have a large effect on the quality of the model.

There are many loss functions available based on a given task: Squared error loss for regression problems, Binary cross-entropy for a two-class classification and categorical cross entropy for many class-classification problem [7]. We will introduce cross entropy loss in this section which is very popular in classification and segmentation tasks [24]. It measures the performance of a classification model whose output is expressed as a probability between 0 and 1. The definition of cross entropy loss for a multi-class classification where $M > 2$:

$$CE(y, \widetilde{y}) = -\sum_{j=1}^{M} y_{o,j} log(\widetilde{y_{o,j}}) \tag{2.3}$$

where y is the actual data or the true distribution and $\widetilde{y}$ is the distribution that network supposes the data follows. $y_{o,j}$ is a binary indicator (0 or 1), if class label *'j'* is the correct classification for observation o, and $\widetilde{y_{o,j}}$ is the predicted probability of observation o is of class label *'j'*.

If $M = 2$, the task is a binary classification, and the binary cross entropy is expressed as [24]:

$$CE(\widetilde{y}, y) = -\frac{1}{M}\sum_{j=1}^{M}\Big[y_j log(\widetilde{y_j}) + (1 - y_j) log(1 - \widetilde{y_j})\Big] \tag{2.4}$$

2.1.6 Activation Functions

In neural networks, activation functions act as transfer functions that map the output of one node to the input of the next node. It is essential to select the best activation functions and understand the effect of the function and its derivative on transforming the data. Three commonly used activation functions are described below.

2.1.7 Sigmoidal Activation Function

The sigmoidal activation function can be expressed by Definition 2.1.1 and the plot illustrating this function can be seen in Figure 2.4.

Definition 2.1.1 (Sigmoidal activation function [25])**.** The sigmodal activation function is given by

$$Sigmoid(z) = \frac{1}{1 + exp(-z)}. \tag{2.5}$$

A good quality about the sigmoidal function is that it is differentiable across its entire domain and, therefore, easy to compute . Also it squashes the input between 0 and 1, which makes it easier to calculate the output as probabilities. Despite this advantage, the sigmoidal function has an issue of vanishing gradient problem that comes from its derivative [8], [25]. The problem arises when the input magnitude to the activation becomes sufficiently large causing the derivative of the function to approach zero as seen in the Definition 2.1.1. This makes the rest of the network to stop learning during backpropagation.

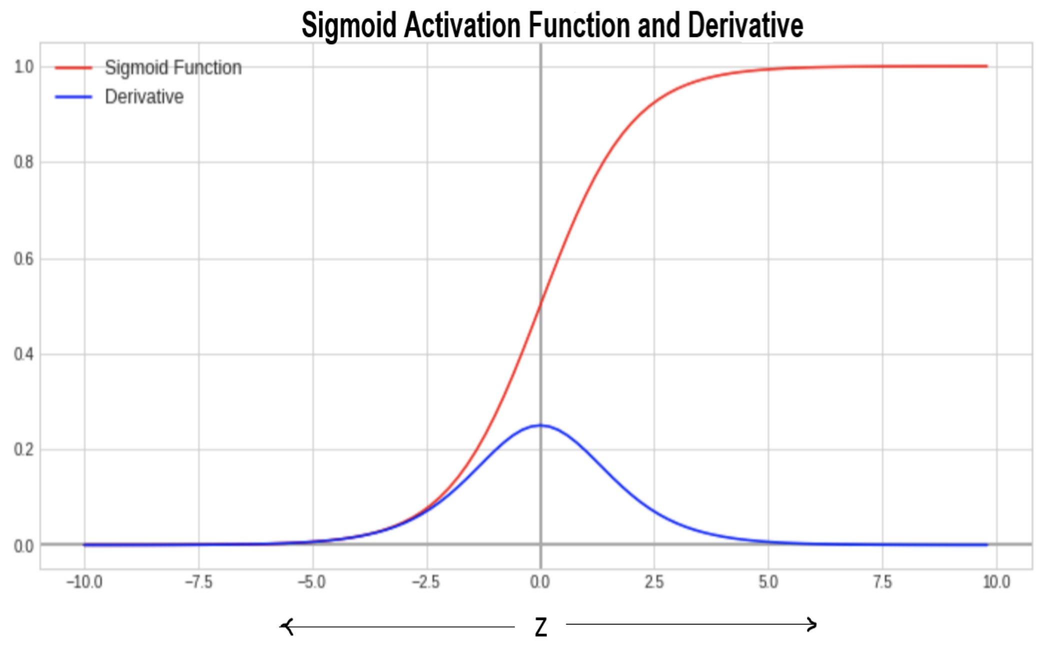


Figure 2.4: Illustration of the Sigmoidal activation function and its derivative [26]

2.1.8 Softmax Activation Function

Definition 2.1.2 (Softmax activation function [25])**.** The softmax activation function is given by

$$Softmax(\boldsymbol{z}_i) = \frac{exp(z_i)}{\sum_{k=1}^{K} exp(z_k)}. \tag{2.6}$$

where z is an input vector to the output layer (if there are ten output units, there are ten elements in vector z). i are the indices of the output units, so i = 1, 2, ..., K.

The softmax activation function normalizes the input value into a vector of values that follows a probability distribution, whose total sums up to one. Usually, all other activations take scalars as input, whereas softmax accepts vectors. Since the outputs are in the range of 0 and 1, it is helpful in accommodating many classes or dimensions in the neural network making it an ideal activation function for multi-class classification problems [25].

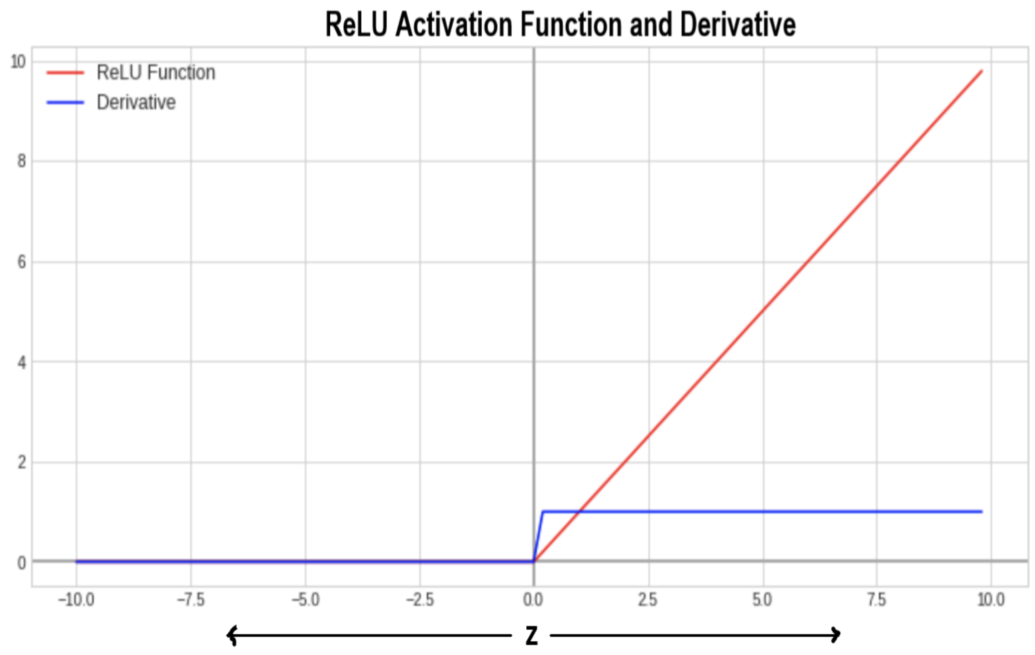


Figure 2.5: Illustration Of ReLU activation function and its derivative [26]
.

2.1.9 ReLU Activation Function

Definition 2.1.3 (ReLU Activation function [27])**.** The ReLU activation is given by

$$ReLU(z) = max(0, z). \tag{2.7}$$

This activation function became very popular due to several properties. Firstly, ReLU solves the problem of vanishing gradient as the derivative of ReLU, as described by Definition 2.1.3, is zero for negative inputs and one for positive inputs as illustrated in Figure 2.5. Secondly, it helps in efficient convergence by outputting more significant update steps. Also, it does not involve any exponential, making it very suitable to compute. All these properties makes it the most used nonlinearity in deep neural networks, [7], [24], [27].

2.1.10 Convolution Neural networks

Convolution neural networks (CNN) are the most famous deep neural network that have shown excellent performance in the field of image classification and semantic segmentation [28]. These networks are capable of extracting localized spatial features that are multi-scaled and are used for performing image analysis. They create a hierarchical structure of features that are formed by the combination of low-level features in a layer-wise pattern to generate high -level features. This can be better understood with images, where low level features like edges and blobs generated from the preceding layer build high-level features like the shape of the different objects present in that image [8]. Also, CNN uses the idea of weight sharing which significantly reduces the need for training a large number of parameters and hence improving the model generalization. Less number of parameters gives easier model training and the model is not prone to over-fitting.

These CNN models have many qualities. Firstly, they have built-in feature extraction in the classification stage and use the learning procedure. Secondly, it is very easy to implement large networks on CNN's as compare to using other artificial neural networks [28], [29].

2.1.11 General model - Convolution neural network

CNN's are gaining popularity because of their quality of image classification based on contextual information [30]. This information describes the shape of an image which produces a better result as compare to pixel-based classification. A key feature discriminating CNN's and ANN's is the convolution layer, where the convolution operation is used instead of matrix multiplication for computing neuron activations. The activations can pass from one layer to the next layer, and the prediction error is backpropagated in the network to update the parameters.

A typical CNN architecture developed by LeCun *et al.* was named LeNet5, shown in the Figure 2.6 [30]. The architecture consists of convolution layers that generate feature maps after convolution operation discussed in section 2.1.12. The feature maps are then downsampled using pooling layers as discussed in section 2.1.13. There is series a of such convolution and pooling layers. The final layer is a fully connected layer as discussed in section 2.1.16 which maps the final output to the desired targets.

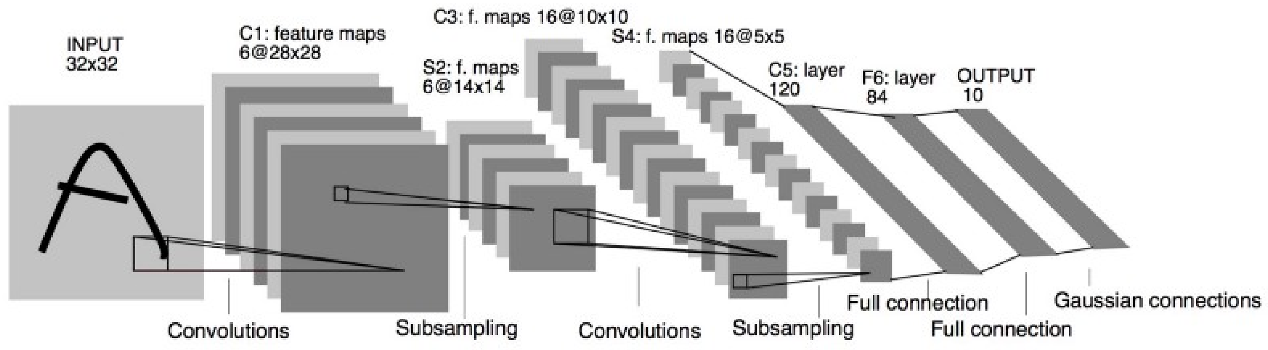


Figure 2.6: Illustration of LeNet-5 CNN architecture used for handwritten character recognition. An image of a character passes through a series of convolution and pooling layers and finally classified using fully connected layers. Here, 16 @ 10 x 10 means 16 filters of size 10 x 10 [30].

2.1.12 Convolution Layer

This layer is the main building block for any convnet (convolution network) architecture. This layer consists of a group of filters that has property of learning different features of an image. These filters are small and spatially oriented along width and height but also have the capability of extending towards full depth in case of 3D input volume [31], [32].

Every element of an activation map comes from a local patch of pixels of the input image known as a local receptive field (Figure 2.7). Receptive field is known as the filter size [8]. The same weights are used across all the patches of the input image. This connection of filters to an input image is local in space along width and height but the filters extend fully to the entire depth of input volume. An example of 3D convolutional operation is described in Figure 2.8 [31].

These filters consists of trainable parameters known as weights of a convolution layer [8], [31]. A single convolution layer comprises of multiple such filters where each filter is small in size as compare to the input volume, but has the same depth or the same number of channels as input provided to the layer. These filters are slid spatially over the height, and the width of the input image and dot products between input and filter are computed at each position spatially during a forward move in the network. This sliding of filters produces a two-dimensional activation map that represents the outputs produced by filters at each spatial position. The network will learn from these filters which get activated when they come across several visual features such as edges of some orientation or color blotch on the first

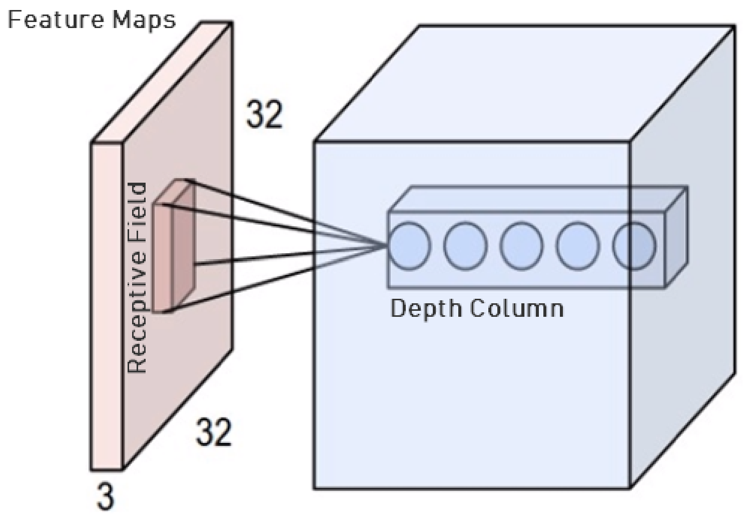


Figure 2.7: Understanding the concept of local receptive field in CNN's. The red box represents the input image of volume (32 x 32 x 3). The blue box represents the convolutional layer where each neuron is connected spatially to a local region of the input volume but across the full depth (i.e., all color channels, 3 in this case). There are five neurons on the depth (the number of filters users want to use), all looking to the same region in the input [31].

layer, which may extend to advanced features like entire honeycomb or wheel-like patterns on higher layers of the network [31].

The response of the filters at the different spatial locations of the input is presented in the feature maps [8]. They also act as a feature identifier where the presence of a given feature indicates a strong response provided by the filter in the receptive field that the filters are observing. These responses can be edges or color changes or anything inside the image that the network assumes to be useful. Choosing the number of filters for a given convolution layer is entirely a design choice by a user designing the interface. The number of filters allows the network to learn more features but also increment the total number of parameters to train. The feature maps outputted by each filter of a given convolution layer are stacked together as a final output and served as input to the next layer of the network. A feature hierarchy is built in any CNN network where it contains multiple convolution layers. During training, filters in the initial layers classify simple features, and the subsequent filters learn complex features of the network. [1]

[1]Deep learning convolution operation consists of filters which are 3D structures and combination of multiple kernels (2D array of weights) stacked together. Kernel is a term used in 2D and filters in 3D.

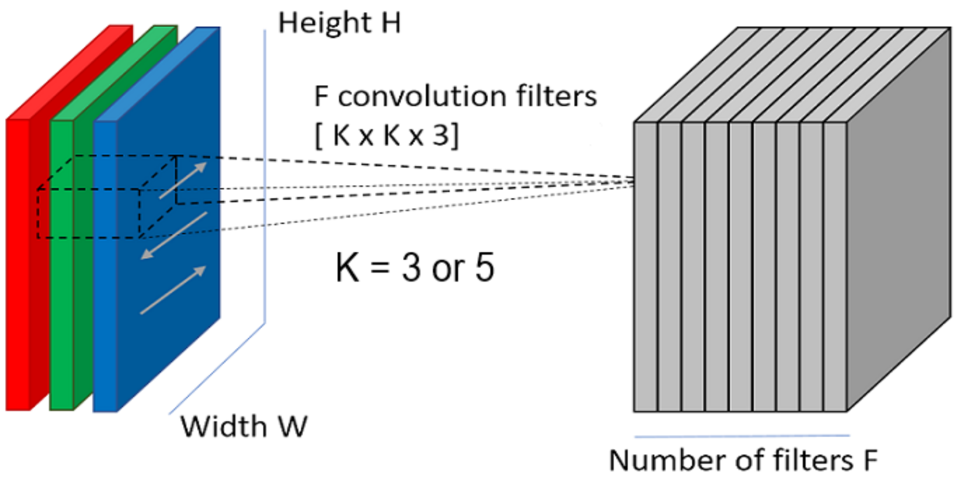


Figure 2.8: Demonstration of 3D convolution operation: An arbitrary 3D RGB image of size H x W x 3. Filters of size 5x5x3 or 3x3x3 are used for convolving input image. A stride of 1 is used for convolving the input by filters and padding = 0. These filters outputs are stacked together to generate a final output which will become an input to the next layer of CNN [33].

The number of trainable parameters of a convolutional layer with n filters, calculated as $(k1 * k2 * C + 1) * n$ where $k1$ is the height, $k2$ is the width, and C is the depth of each filter. The shape of the output of the convolutional layer is controlled by three hyper-parameters: depth, stride, and padding [31]. The depth of the output determines the number of filters used in the convolutional layer, where each filter is trying to find some new feature in the input image.

Stride helps in controlling the step size with which the filter convolved over the input image [8], [34]. Consider setting the stride to 1, the filters will then move by one pixel at a time. If stride is set to 2 (or more than 2 which is very uncommon in practice) the filter will slide over the image by two pixels horizontally and vertically at a time. If the stride becomes equal to the filter size then every pixel in the image will be used once by each filter. This will cause fewer border pixels participating in convolutions as compared to pixels closer to the center [31]. It will eventually cause loss of information contained by the border pixels.

Increasing the stride decreases the output size when passed through a large number of convolution layers causing loss of information. For preserving the information contained at the border pixels and output spatial dimensions, input images are padded with zeros along the borders which is known as **Zero-Padding** [8], [34].

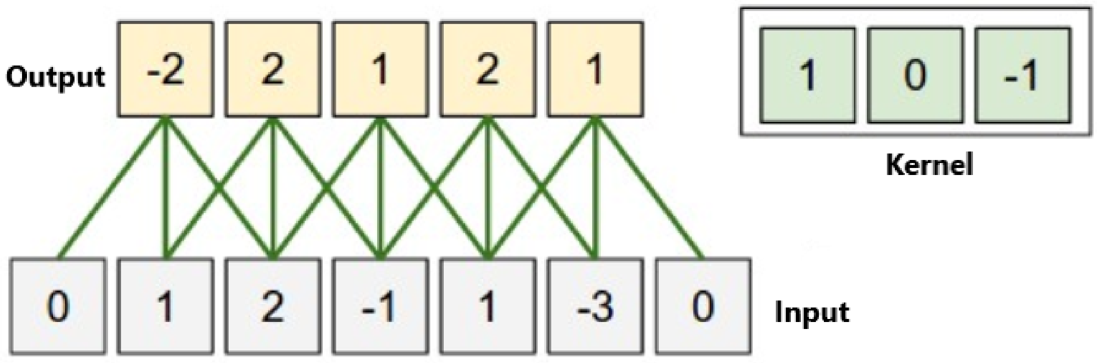


Figure 2.9: Illustration of Zero-Padding with 1D convolution. Input size is n = 5, zero-padding p = 1, stride = 1 and kernel (1, 0, -1) of size m = 3. The output size is calculated by convolution output formula as shown in eqation 2.1.12 : ((5 + 2 * 1 - 3) / 1) +1 = 5. The output (-2, 2, 1, 2, 1) size after convolution is same as input size, this type of padding operation is known as *same padding*. The green line represents the convolution operation when the kernel is slided over the input and the output is generated with a stride of 1. For example when kernel (1, 0, -1) is slided over input region (0, 1 ,2) the output value obtained is -2. The kernel is then move to next set of pixels by a stride of 1 [31].

The effect of zero padding can be seen in the Figure 2.9.

Calculating the size of the convolution output

The size of the output obtained after a convolution operation, can be calculated by obtaining the total frequency with which the filter has moved along the input. Assuming an input vector of size n and size of filter as m. The output size resulted from convolution operation with padding p and stride s can be calculated using the below formula [8]:

$$output = \left\lfloor \frac{n + 2 * p - m}{s} \right\rfloor + 1 \tag{2.8}$$

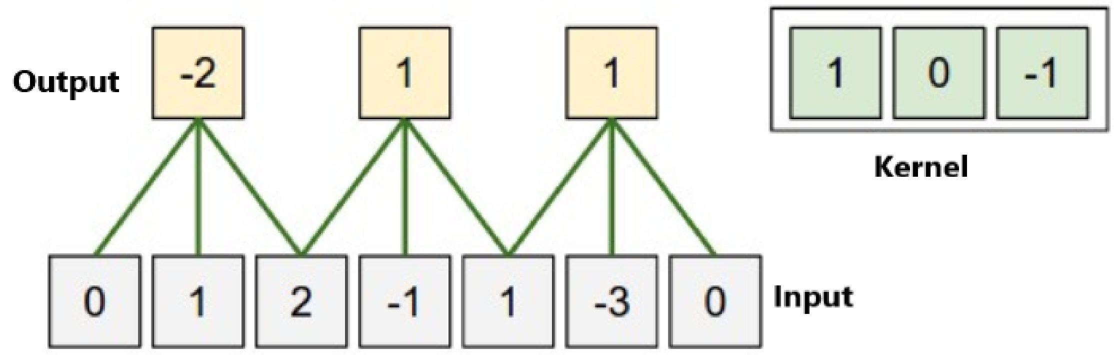


Figure 2.10: Illustration of 1D strided-convolution with zero-padding. The input size n = 5, zero-padding p = 1, stride = 2 and kernel (1, 0, -1) of size m = 3. The final output size is calculated by convolution output formula as described in equation 2.1.12 : ((5 + 2 * 1 - 3) / 2) +1 = 3. The output (-2, 1, 1) size after convolution is smaller as input size, this type of padding operation is known as *valid padding.* The green line represents the convolution operation when the kernel is slided over the input and the output is generated with a stride of 2. For example when kernel (1, 0, -1) is slided over input region (0, 1 ,2) the output value obtained is -2. Then the kernel is moved 2 pixels ahead as stride is 2 and perform the convolutional operation so obtain the next output i.e. 1 [31].

2.1.13 Downsampling operations

The filters or the feature detectors in a convolutional neural network should have a large receptive field so that the network can recognize the feature that spans more in the input. With small filter sizes the receptive field increases slowly. If we want to increase this field, the number of layers in the network also increases which in turn increases the number of trainable parameters of the network. It is therefore very important to find some techniques that increase the receptive field without increasing the number of parameters in the network. The technique that works best for this is down-sampling operations [8] ,[24]. There are several ways to down-sample the input in the CNN network such as strided convolutions, Max-pooling and Average-Pooling.

Strided convolution generally refers to convolutions operations having stride greater than 1. Figure 2.10 shows the operation of 1D strided convolution with a stride of 2 and convolutions on an input of size 5 , with padding 1 and receptive field (1, 0, -1) size is 3. In such strided convolution using stride as 2, the dimension of the output feature map is downsampled and the size of output becomes 3 [31].

Pooling Layer

Downsampling operations prove to be very beneficial in convolutional neural networks. As discussed above, Strided convolutions are easy to implement, but there are more efficient methods available for downsampling such as Max-pooling and Average-Pooling [35], [8].

Max-pooling layer in CNN performs max-pooling operation where windows select from the input feature maps, and the maximum value of each channel is yield as output [36]. This operation is similar to convolution conceptually, except for the fact that in convolution, the local patches from the input are transform using a linear transformation (using the convolutional kernel) [31].

In contrast, in pooling, these patches are processed using a max- tensor operation that is hardcoded. It is a kind of action where there is a dimension reduction or downsampling of the input, reducing the number of feature map coefficients that need to be processed by the network. This layer consist of no trainable parameters (Figure 2.11). There are plenty of other pooling operations available [24], and those will not be discussed here, as strided convolution and max-pooling are the methods commonly used for downsampling. Max-pooling generally gives better results as compare to strides. Strides are used because of their easy implementation and simplicity [7], [35].

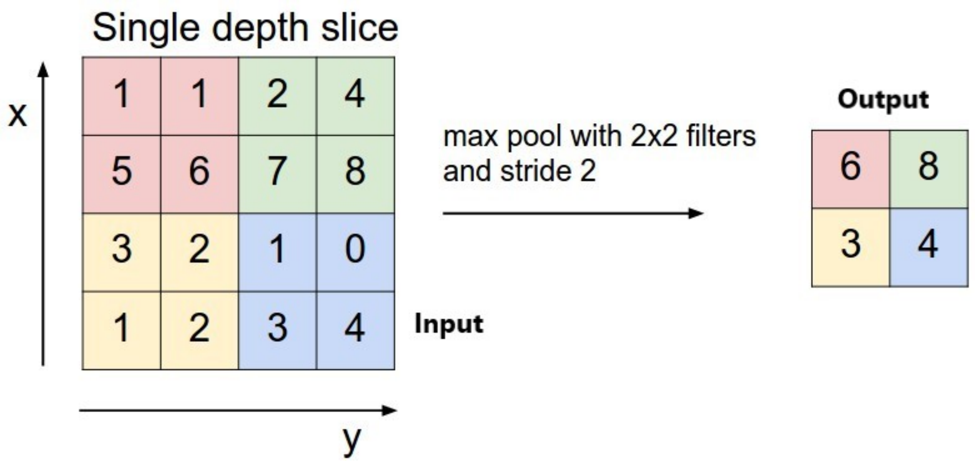


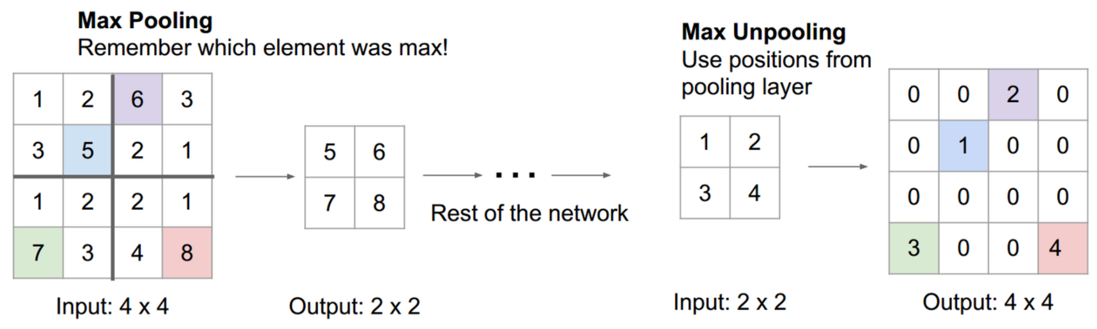
Figure 2.11: A max-pooling operation with filter size 2x2 and stride 2 is used during convolution operation. The Max-pooling operation down-sampled 4x4 input to a 2x2 output matrix. Each element in the output corresponds to the largest value in the corresponding quadrant of the input , for example consider the red box entries (1, 1, 5, 6) in the input, the max pooling operation will select the maximum entry (6) and store this value in the output [31].

2.1.14 Upsampling Operation (Transposed Convolution)

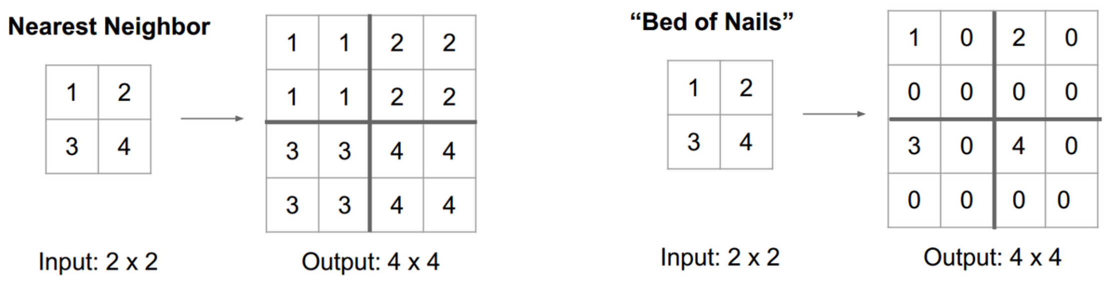
Neural networks, used for generating images, involve upsampling of the generated feature maps from lower resolution to higher resolution [7], [37]. This upsampling plays a significant role in semantic segmentation networks where the user wants to obtain the output segmented image to have the same dimension as the input image. The Figure 2.12 illustrates few methods of upsampling the feature maps that do not require any trainable parameters and are only dependent on the factor of upsampling and content in the feature map.

The other option of upsampling is known as Transpose convolution or fractionally strided convolution, that does not require any trainable parameters [37] ,[38]. In this type of convolution, the upsampled feature map are generated by a periodic shuffling of several intermediate feature maps that are created by applying multiple convolution operations on the input feature maps.

Equation 2.9 illustrates the convolution operation where a 3 x 3 input matrix is flattened into a column vector of size (9,1) denoted by the vector $[x1, x2, \cdots, x9]$. The kernel $(w1, w2, w3, w4)$ is of size 2 x 2 , rearranged in the form of convolution matrix C (4,9). Each row of C determines a convolution operation. Now, a matrix multiplication operation of convolution matrix C (4,9) with column input



(a)



(b)

Figure 2.12: An illustration of different techniques of upsampling. Figure (a) Down-sampling the input using the max pooling operation similar to shown in figure Figure 2.11 and corresponding upsampling of input using Max unpooling where the de-pooled area of the output is filled with the maximum element position in the max-pooling operation and remaining elements are set to zero. (b) In nearest neighbours upsampling, the nearest neighbor goes to the pooled output by filling in the de-pooled area with the current element of the input and in bed of nails technique the input element is arranged in the upper left corner of the de-pooling area, and other elements are set to zero. Both these techniques do not require any trainable parameters **CNN:convolutionallayer.**

vector (9,1) generates an output $(z1, z2, z3, z4)$ of shape (4,1). The output is then rearranged as the final output matrix of size (2 x 2). The input (3 x 3) is downsampled to the output of size (2 x 2) by convolution operation.

$$C \begin{bmatrix} w1 & w2 & 0 & w3 & w4 & 0 & 0 & 0 & 0 \\ 0 & w2 & 0 & w3 & w4 & 0 & 0 & 0 & \\ 0 & 0 & 0 & w1 & w2 & 0 & w3 & w4 & 0 \\ 0 & 0 & 0 & 0 & w1 & w2 & 0 & w3 & w4 \end{bmatrix} \begin{bmatrix} x1 \\ x2 \\ x3 \\ x4 \\ x5 \\ x6 \\ x7 \\ x8 \\ x9 \end{bmatrix} = \begin{bmatrix} z1 \\ z2 \\ z3 \\ z4 \end{bmatrix} \tag{2.9}$$

Equation 2.10 illustrates a transpose convolution operation where an input of size (2 x 2) will be upsampled to (3 x 3) using transpose matrix C^T. Here a kernel with weights $(c1, c2, c3, c4)$ of size (2 x 2) is arranged as a transpose convolution matrix C^T (9,4). The input size is (2 x 2), where $(z1, z2, z3, z4)$ denotes the entries of input arranged as a column vector (4,1). The matrix multiplication between input vector and C^T will obtain an output vector (9,1) denoted as $(x1, x2, \cdots, x9)$,which is rearranged to obtain the final output matrix of size (3 x 3).

$$C^T \begin{bmatrix} c1 & 0 & 0 & 0 \\ c2 & c1 & 0 & 0 \\ 0 & c2 & 0 & 0 \\ c3 & 0 & c1 & 0 \\ c4 & c3 & c2 & c1 \\ 0 & c4 & 0 & c2 \\ 0 & 0 & c3 & 0 \\ 0 & 0 & c4 & c3 \\ 0 & 0 & 0 & c4 \end{bmatrix} \begin{bmatrix} z1 \\ z2 \\ z3 \\ z4 \end{bmatrix} = \begin{bmatrix} x1 \\ x2 \\ x3 \\ x4 \\ x5 \\ x6 \\ x7 \\ x8 \\ x9 \end{bmatrix} \tag{2.10}$$

The main thing to keep in mind while using transpose convolution is the connectivity pattern used in C^T. There is an association between the input and the output, which is handled in the backward direction as compared to standard convolution matrix C (one-to-many relationship as compared to many-to-one in regular convolution operation) [38], also we can not take convolution matrix C and use the transpose as transpose convolution matrix. That is the reason the entries of C and C^T are different.

2.1.15 Batch Normalization

Batch Normalization (BN) is a widely selected procedure that helps in fast and stable training of deep neural networks [39]. The effectiveness of why the batch normalization works is not entirely understood, but still, it has been used in most of the modern neural network architectures [40]. This technique is introduced by Ioffe and Szegedy [39] for accelerating the training process of deep neural network by standardizing the inputs for each layer and therefore reducing the internal covariance shift.

Reduction of *Internal covariance shift* [39] is a normalization step that helps in fixing means and standard deviation of inputs before feeding it to the next layer. This is achieved by subtracting the batch mean from input and dividing by the batch standard deviation. It will help a model to have high training speed as the output is linearly transformed, having zero mean and unit standard deviation. BN is applied after convolution and before introducing non-linearity in the layer [41].

Assuming the output of a hidden layer X is a matrix of dimension (N, D), where N is the number of input samples present in the batch and D is the number of hidden units, the first step is normalizing X as shown below :

$$\widetilde{X} = \frac{X - \mu_b}{\sqrt{\sigma_B^2 + \epsilon}} \tag{2.11}$$

where μ_b is the mean of the input batch, σ_B^2 is the standard deviation and ϵ is a constant added to the standard deviation for numerical stability to avoid division by 0. As it is clear from the above equation, μ_b and σ_B^2 are differentiable, making $\widetilde{X}$ also differentiable. The linear transformation of y is given by:

$$y = \gamma\widetilde{X} + \beta = BN_{\gamma\beta}(X) \tag{2.12}$$

Then we refer to this transform as *Batch Normalizing Transform* represented by $BN_{\gamma\beta}(X)$. γ and β are learnable parameters of the network. The BN transform can be added to a network for transforming the output of activation layer.

There are many advantages of using batch normalization. One among them is tackling the problem of exploding or vanishing gradient. The reason for exploding gradients in the deep neural network is due to the high learning rate that makes the network stuck in local minimum [39]. BN solves this issue of vanishing gradient by transforming input layers into a form that ensures zero mean and unit variance. Also, it makes the network more robust as we get layers that are not affected by the scale of the parameters and hence solving the issue of exploding gradient.

2.1.16 Fully Connected Layer

The convolutional neural network used for classification problems consists of a series of convolution, activation, and pooling layers that are followed by a couple of fully connected layers, as shown in the Figure 2.6. A set of feature maps are created by a series of convolution and pooling operations. The feature maps describe information about the high-level features present in an input image (out of a collection of learned features that the network considers useful for separating them into different classes available in the datasets). The fully connected layers used the feature maps and flattened them into a vector of class probabilities [8], [31].

There are two main downsides of using a fully connected layer. Firstly, a fully connected layer has (n x m) parameters, which will generate a large number of parameters for most layers, leading to high memory usage and overfitting [42]. Therefore networks using a fully connected layer are not feasible for image processing tasks.

Secondly, all spatial information about the location of the feature is lost due to the use of dense layers. It is not a concern if the purpose is to predict a class label for an entire image. However, for the network that is performing semantic segmentation, this will create an issue, as the aim of the model will be pixel-wise dense predictions, i.e., to predict a class label for each pixel of the image.

2.1.17 Residual Connections

The depth (number of layers) of a CNN network has a significant impact on its performance [43]. More layers are usually more beneficial as it enables a system to create a rich feature hierarchy. The problem with going deeper in a network is that the gradients have to propagate through more number of layers while training, and that will lead to a vanishing gradient problem. The vanishing gradient problem can be decreased with the use of batch normalization but can not be entirely solved by it.

Generally, increasing the depth of a network should improve the accuracy, but in some cases, deep networks have resulted in worse performance than their counterpart shallower CNN's. For example, consider a shallow CNN performing at some level. Let us increase the model depth by k layers. There is an expectation that the deeper network will perform better than the shallow system as it learns identity mapping while training. However, in practice, this does not happen, as the deeper network faces difficulties in learning identity mappings, which leads to the problem of degradation. To solve this, He *et al.* proposed the concept of residual connections, also known as short skip-connections [44].

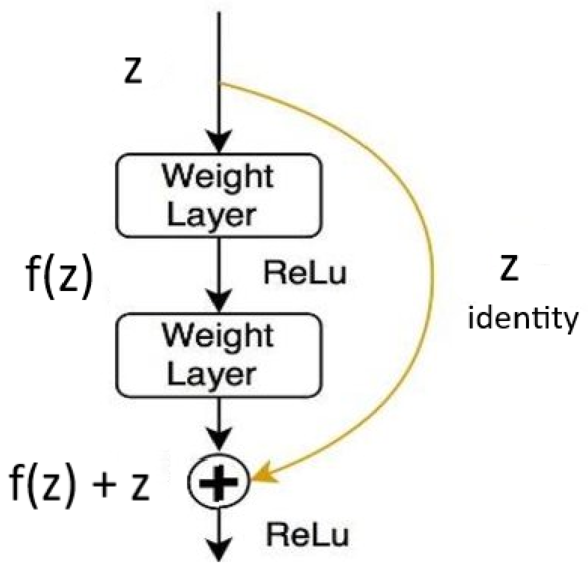


Figure 2.13: Illustration of residual connection. A weight layer denotes a layer that modifies the input (for example, a convolution layer). A residual connection, z, skips the weight layers and adds the unmodified information to the output of weight layers [44].

Residual-connections allow information to skip one or more layers in the CNN network. The Figure 2.13 illustrates how residual connections implemented using identity mapping. The output feature map obtained from the skip connections is added to the output feature maps resulted from the stacked convolutional layers to generate a final output y, as shown in the below equation :

$$\boldsymbol{y} = \boldsymbol{f}^{(res)}(\boldsymbol{z}) + \boldsymbol{z} \tag{2.13}$$

Here, $f^{(res)}$ is the residual function and z is the input which is being fed to the model. There is one problem related to the formula stated above. It will not operate if output dimension changes after a convolution operation, either with the use of strides or with a change in the number of channels. Instead, an approximate identity map is learned by the network which will modify the Equation (2.13) as,

$$\boldsymbol{y} = \boldsymbol{f}^{(res)}(\boldsymbol{z}) + id(\boldsymbol{z}) \tag{2.14}$$

here, id is a function that approximates the identity function. It is known as identity map if the output dimension is equal as input and it permits in constructing deeper CNN's network.

2.1.18 Regularization

Regularization in deep neural network [7], refers to a set of different methods that help in lowering the complexity of the model while training, such that the model generalizes better and hence prevent over-fitting. It also improves model performance when the model is run on unseen data.

One technique of reducing over-fitting is to reduce the complexity of the network by making the weights take only smaller values making their distribution more regular. This technique is known as weight regularization, which is achieved by adding a cost associated with having larger weights to the loss function. One such method of transforming the loss function is known as L_2 *regularisation*, also known as weight decay in neural networks. Mathematically it is equivalent to making the gradient small [24], [7].

The loss function in L_2 *regularisation* is modified as shown below,

$$\tilde{J} = J + \alpha ||W||_2^2, \tag{2.15}$$

Here, J is the original loss function, $\tilde{J}$ is the modified loss function, α is the parameter describing the amount of regularisation, $||W||_2^2$ is the sum of squared weights. Both together are known as the regularization penalty term added to the cost function for encouraging lower weights and hence introducing stability for input [7].

One more popular method for regularizing a deep neural network is known as early stopping [24]. An issue while training neural networks is the choice of the specified number of iterations to use for training, where a large number of iterations can cause overfitting, and less can cause underfitting. Early stopping therefore, can help in stopping the optimization early so that the network has very little time to overfit the training data .

Finally, we will discuss the most effective and most commonly used method for regularization: *Dropout* [45]. Dropout helps in preventing overfitting and provides an efficient way of exponentially combining different neural network architectures. The term 'Dropout' introduced by Srivastava *et al.*, is an averaging technique based on randomly dropping some units (visible and hidden) during training, which may be input data points to a layer or activation's from the previous layer. These units are dropped temporarily from the network along with all their incoming and outgoing connections.

During every iteration of training, a fraction of hidden units are randomly dropped with a probability as shown in Figure 2.14, The dropout probability p_{drop}(or the keep probability p_{keep}= 1-p_{drop})[8], which is known as dropout rate is usually set between 0.2 and 0.5. Weights of the remaining neuron are re-scaled for accounting for the missing units [7], [24].

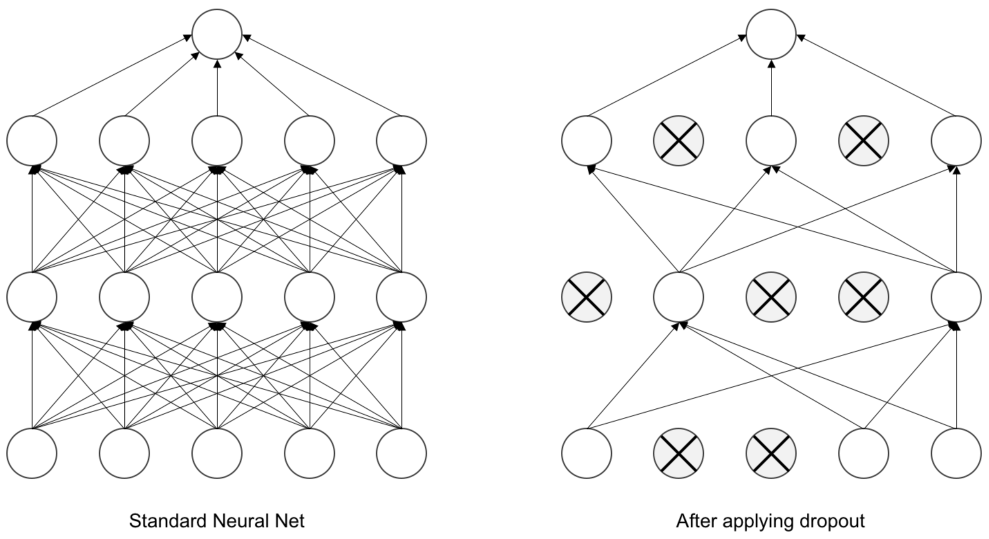


Figure 2.14: Illustration of Dropout neural net model, **Left** :A neural network with two hidden layers. **Right** :A neural net obtained using Dropout. The circle with "X" denotes the neurons or nodes of the network that are randomly dropped during training [45]. The dropout rate chosen to be 0.5.

For example, during training, consider for a given input, a layer is giving an output vector [0.3, 0.4, 1.3, 1.8, 0.4]. After applying dropout to this layer, the output will have some zero entries at random, which will produce an output vector [0.3, 0, 1.3, 1.8, 0]. This dropout helps the network to learn redundant representations of data.

During testing, no units are dropped, and all the layer's output values are scaled down by a dropout rate factor to compensate for the fact that more units are active as compared to the training phase [7]. For example in Figure 2.14 consider, that during training if we drop 50% of the units in the output i.e. the dropout rate is 0.5 then at test time, we will scale down the output by the dropout rate. So the layer output will become 0.5 * layer_output.

2.1.19 Optimization

Gradient Descent

One of the most popular algorithms used for performing optimization is Gradient Descent and a standard method of optimizing neural networks. This a way to find the global-local minimum of the objective function, which helps in exploring the weights and biases that result in lowering the loss and giving the most accurate predictions [8], [24].

This derivative of the loss function is known as the gradient . In every iteration, steps are taken in a direction opposite to gradient descent where the step size in determined by the learning rate and slope of the gradient. This process continues until the metric used to estimate performance reaches a predetermined value, or there is no performance improvement in the network.

Using the concept of *Gradient Descent* weights are updated by taking steps in the opposite direction of the gradient ($\nabla J(w^{(t)})$) of our cost function $J(w^{(t)})$.

The weight update using gradient descent :

$$w^{t+1} = w^t + \Delta w^t \tag{2.16}$$

Here, the weight change Δw^t is calculated as the negative gradient multiplied by the learning rate η:

$$\Delta w^t = -\eta^{(t)} \nabla J(w^{(t)}) \tag{2.17}$$

Here, $w^{(t)}$ are the weight parameters at time step t. J is the loss we want to minimise.

The above equation causes problems of using gradient descent with large datasets in neural networks. Calculating the derivative of the cost function and looping through the entire dataset for each step of gradient descent requires high computational power and is very expensive.

There is a solution to solve the above problem, where gradient ∇J is replaced with a random variable ∇J_{rand} with the following property

$$\mathbb{E}[\nabla J_{random}] = \nabla J. \tag{2.18}$$

Here, $\mathbb{E}$ represents the expected value of J_{random} which is equivalent to the loss. The algorithm used for finding the gradient of the loss using a random variable ∇J_{random} is called *stochastic gradient descent or SGD* [46], [24], which is the most popular optimization algorithm used in deep neural networks. Generally, we choose ∇J_{random} to be equal to

$$\nabla J_{random} = \sum_{\boldsymbol{x},\boldsymbol{y}_i \in \mathcal{C}^{(i)}} \nabla J(w^{(t)}; \boldsymbol{x}_i, \boldsymbol{y}_i) \tag{2.19}$$

This algorithm performs parameter update on each training example x_i and their labels y_i such that the gradient of the loss ∇J_{random} is now chosen from $\mathcal{C}^i$, which is a small random subset of the training set.

This random subset of data is generally chosen with a without replacement technique. To understand this, if during an iteration one data-point is selected from the whole dataset, then this data-point will not be chosen in the following iteration.

One drawback of using SGD is that it does not converge fast [24], [46]. An oscillating behavior of the gradient occurs due to choosing the direction of steepest descent as shown in Figure 2.15, which can be overcome by using other algorithms such as SGD with momentum and Adam [47]. There are more algorithms present, but we will discuss the above two.

Momentum Gradient Descent

Momentum gradient descent is one method of reducing the oscillations in SGD [49]. Momentum can be defined as the average of the gradients, which is then use

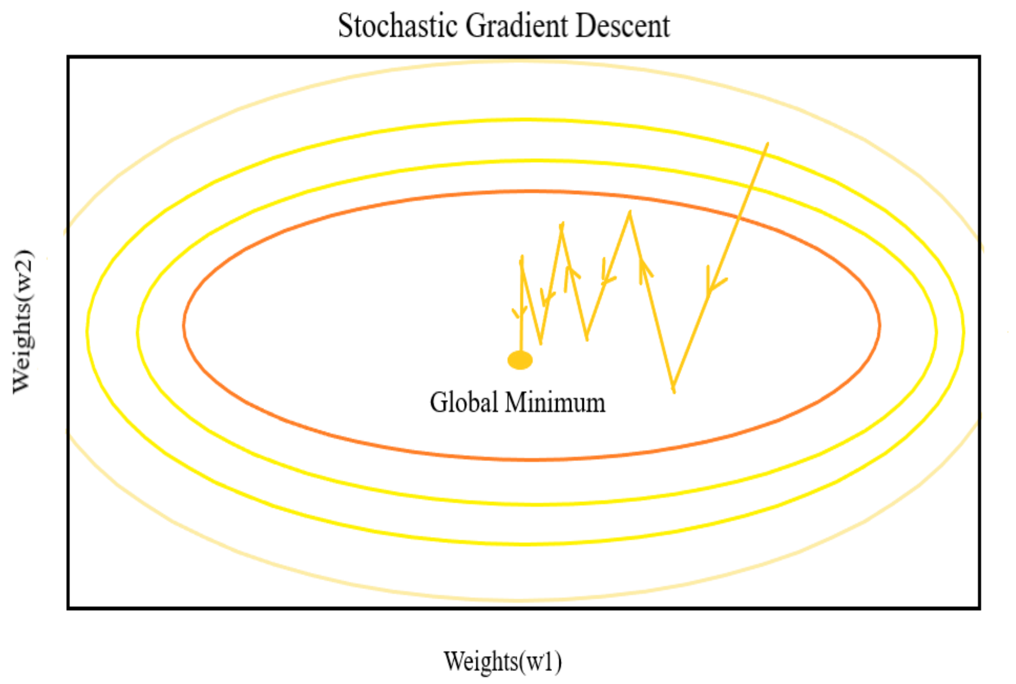


Figure 2.15: Demonstration of choosing a direction of steepest descent by the gradient that lead to oscillations and hence reduction in convergence speed. The orange path shows the direction followed by the SGD optimizer with too large learning rate. The ellipses are level curves of quadratic loss [48].

to update the weight of the network. This can be expressed as follows:

$$\boldsymbol{V}^t = \beta \boldsymbol{V}^{t-1} + \eta \nabla J(w^t); \boldsymbol{x}_i, \boldsymbol{y}_i) \tag{2.20}$$

Here, $\beta \in (0, 1)$ acts as a hyperparameter, which effectively replaces the gradient by the one that has averaged over multiple past gradients. $\boldsymbol{V}$ is known as the momentum (its typical value is about 0.9). The value of $\boldsymbol{V}$ incorporates past gradients similar to how a ball rolling down the loss function landscape integrates over past forces.

And the final update of weights using momentum gradient descent can be calculated as:

$$w^{t+1} = w^t - \boldsymbol{V}^t \tag{2.21}$$

The notion behind momentum is understood by considering the optimization process as a small ball rolling down in the direction of the loss curve. If the ball has sufficient momentum, it will not be stuck at the ravine or the local minimum and can reach the global minimum. In this situation, momentum implemented by

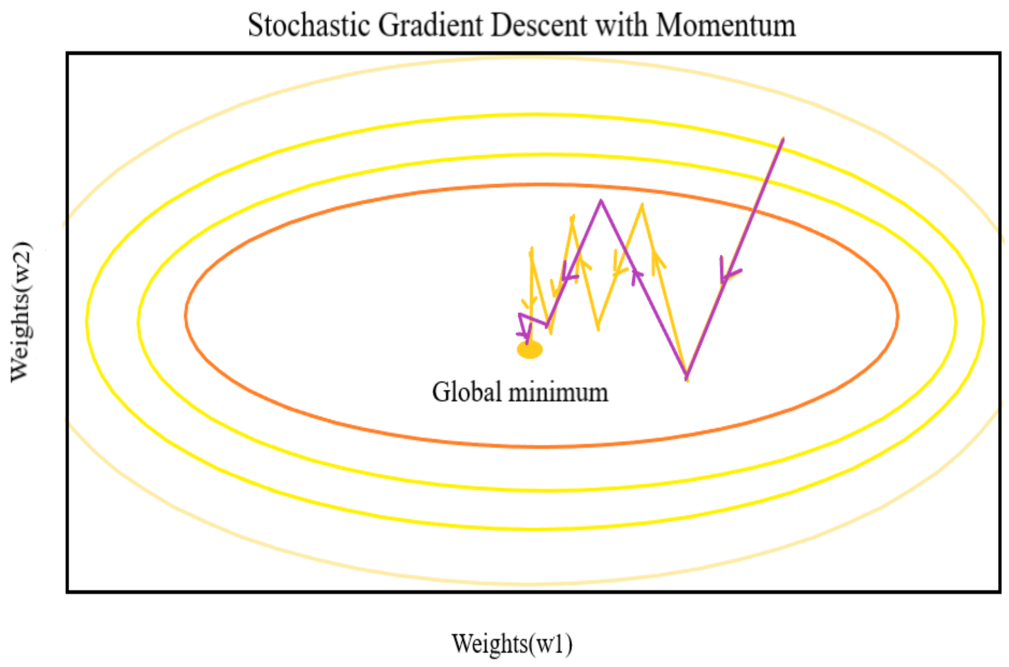


Figure 2.16: Demonstration of stochastic gradient descent with momentum. The orange arrows corresponds to the path followed by the SGD and the purple arrows correspond do the path followed by momentum gradient descent.The ellipses are level curves of quadratic loss [48].

moving the ball at each step is based on the current slope (current acceleration) value as well as on the current velocity (resulted from past acceleration) [7], [49]. This is shown in Figure 2.16.

Adaptive Moment Estimation(Adam)

Adam is an algorithm [47], [49], used to enhance momentum gradient descent and invented by Kingma and Ba. This method calculates the adaptive learning rate for each parameter from estimates of the first and second moments of the gradient. This algorithm is represented as:

$$\boldsymbol{m}^{(t)} = (1-\beta_1)\nabla J_{rand}(w^{(t)}) + \beta_1 \boldsymbol{m}^{(t-1)} \tag{2.22}$$

$$\boldsymbol{v}^{(t)} = (1-\beta_2)\left(\nabla J_{rand}(w^{(t)})\right)^2 + \beta_2 \boldsymbol{v}^{(t-1)} \tag{2.23}$$

Here, $\boldsymbol{m}^{(t)}$ is the estimates of the first moment (the mean)and $\boldsymbol{v}^{(t)}$ is the second moment (the uncentered variance) of the gradients. Also, $\beta_1, \beta_2 \in (0,1)$ are the hyper-parameters which helps in controlling the decay rates of the moving averages ($\boldsymbol{m}^{(t)}$ and $\boldsymbol{v}^{(t)}$). $\nabla J_{rand}(w^{(t)})$ is the gradient with respect to weight parameters at time step t.

The $\boldsymbol{v}$ term in Adam differentiates it with momentum gradient descent. The thought behind using this term is that if a parameter is facing large updates than the previous iterations, then we can decrease the learning rate. This reduction in learning rates is generally related to numerical instabilities. The illustration of this phenomenon is represented in Figure 2.15.

The moving averages ($\boldsymbol{m}^{(t)}$ and $\boldsymbol{v}^{(t)}$) are initialised as zero, leading vectors m and v biased towards zero during the initial time steps, and mainly when the decay rates are low (i.e., β_1, β_2 are approaching one).

To fix this initialization issue, Kingma and Ba created the terms $\boldsymbol{m}^{(t)}$ and $\boldsymbol{v}^{(t)}$:

$$\hat{\boldsymbol{m}}^{(t)} = \frac{\boldsymbol{m}^{(t)}}{1-\beta_1^t} \tag{2.24}$$

$$\hat{\boldsymbol{v}}^{(t)} = \frac{\boldsymbol{v}^{(t)}}{1-\beta_2^t} \tag{2.25}$$

The weight parameters are finally updated using the Adam algorithm as shown below:

$$w^{(t+1)} = w^{(t)} - \eta^{(t)} \frac{\hat{\boldsymbol{m}}^{(t+1)}}{\sqrt{\hat{\boldsymbol{v}}^{(t+1)}} + \epsilon} \tag{2.26}$$

Here, η is the learning rate, and ϵ is used to give numerical stability [47], hence to prevent division by zero.

The advantage of using Adam is that it is extremely fast [47]. But the drawbacks associated with using Adam is that the generalization properties are not good as compare to SGD and momentum SGD [50] while training different models. So, the model trained using Adam will sometimes perform poorly on unseen data that is not used while training the model. But it is still in use as an optimizer, as it saves time during training and helps the user in modifying other model design related tasks such as hyperparameter tuning.

2.1.20 Semantic Image segmentation using convolutional neural network

Semantic image segmentation has become one of the essential applications in image processing and computer vision domain, which is widely used in the medical imaging field [51]. It is a method of grouping parts of the image together that belong to the same object class.

A typical semantic segmentation network consists of an encoder-decoder architecture shown in Figure 5.3, use to delineate the boundary of the tumor in a brain MRI image [52]. The higher-level features are produced by the encoder using convolution, and these features are interpreted by the decoder using the class targets. The encoder uses max-pooling layers as discussed in section 2.1.13 to decrease the spatial dimension gradually, and the decoder will slowly recover the spatial dimension and object details using transpose convolution operation as discussed in section 2.1.14. There are several such encoder-decoder architectures available for semantic segmentation, but we will discuss the two popular designs mainly used in medical image analysis: U-Net [53] and V-Net [23].

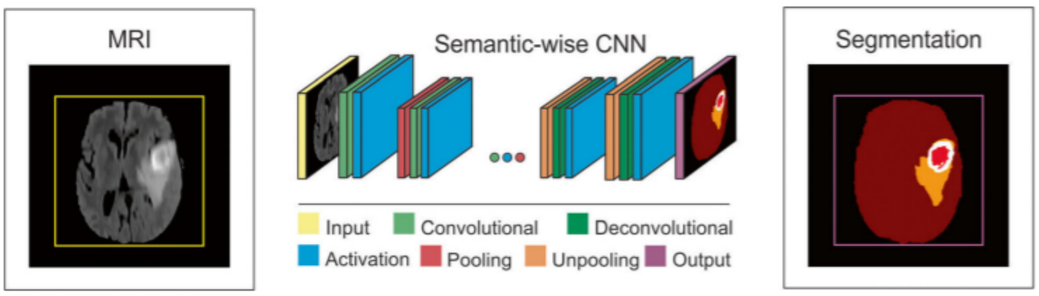


Figure 2.17: Illustration of semantic-wise CNN architecture used for segmenting brain tumor from a brain MRI image [52].

2.1.21 Architectures for semantic image segmentation

U-Net

U-Net is a well known encoder-decoder architecture proposed by Ronneberger et al. [54], which is used for semantic image segmentation. It consists of the contracting path and an expansive path, which are applied during image object detection. Figure 2.18 is an illustration of the U-net architecture. A limited amount of image data in any medical image analysis task is a big challenge for better performance. U-Net helps in utilizing the annotated data samples more efficiently and hence reduces the need for a larger dataset [53].

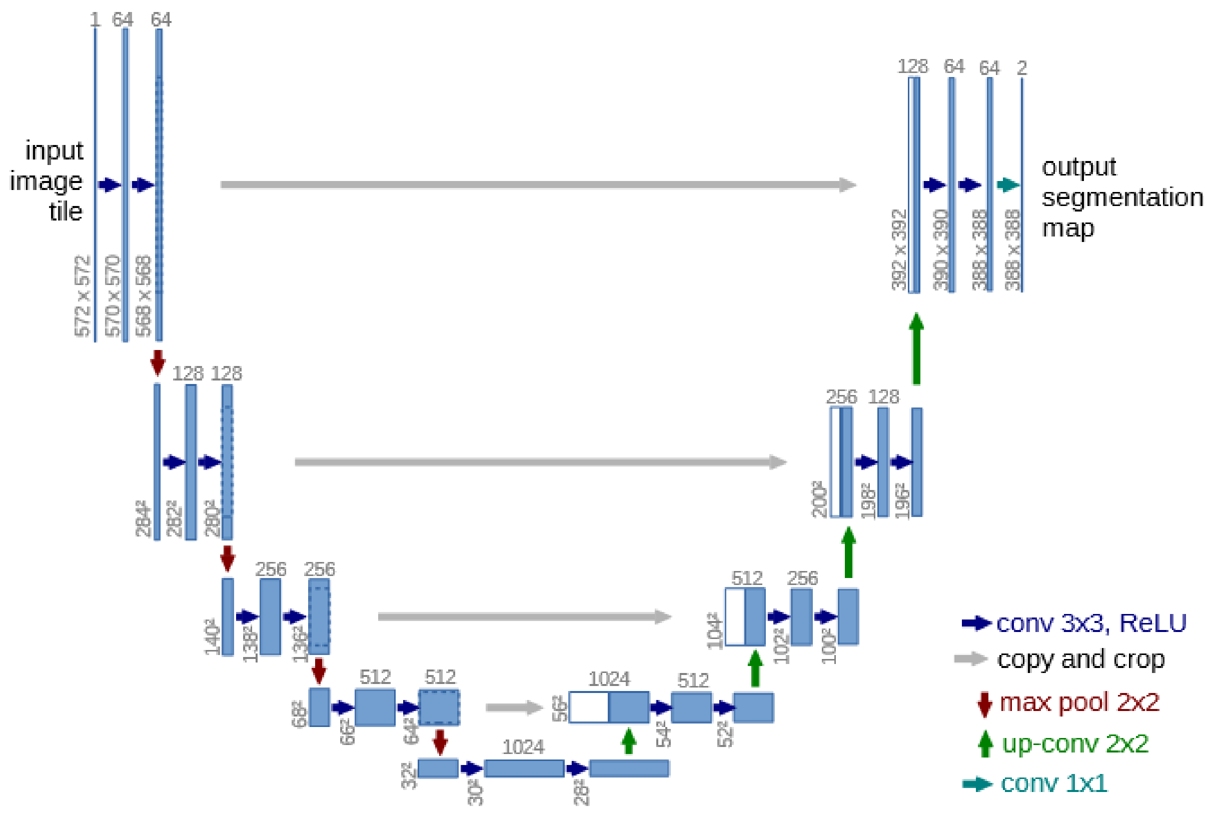


Figure 2.18: Illustration of U-net architecture (example for 32x32 pixels in the lowest resolution). Blue boxes represent feature map with multiple channels. The number of channels is written on top of each box. The dimension of image is written at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations [54]. The blue arrow signifies the convolution operation with applying ReLU non-linearity, gray arrow describes the skip connections for concatenating the features from the contracting path to the expansive path for preserving the image pixel information. Red arrow is for max-pooling operations , green arrow for up-convolution operations and, finally the light blue arrow is the final 1 x 1 convolution operation.

The contracting path follows the architecture of a convolutional network, having repeated operation of 3x3 convolutions (unpadded convolutions). Non-linearity can be applied using a non-linear activation function, mainly a rectified linear unit

(ReLU) max(0,x) as discussed in section 2.1.3 [27]. For further downsampling of the image, max-pooling operation with stride two is executed on the output feature map. Each step of down-sampling doubles the number of feature channels.

In the decoding or expansive path, 2x2 up convolutions are applied that decrease the number of available feature channels to half. This path hence up-samples the image dimension.

Max-pooling in the contraction path helps in achieving high translation invariance with small shifts spatially in the input image. Several layers of max -pooling causes a loss of spatial resolution of the feature maps. There is a high-frequency information loss around the borders of the image, which is not beneficial for any segmentation task where boundary delineation plays a vital role. This loss is reduced by using long -distance skip connections introduced in the U-net architecture[53]. Long-distance skip connections signify that cropped feature maps from the contracting path are concatenated correspondingly in the respective feature maps of the expansive path to preserve the border pixel information as shown in Figure 2.18 by the gray arrows, which is then followed by a ReLU.

In the last layer, a 1x1 convolution is applied to map all the components in the feature vector to the desired class labels. There are 23 convolutional layers in total used in the model [53], [54]. This architecture can help in processing 2D image data.

V-Net

Medical image analysis mostly uses data that consist of three dimensional image data. We will illustrate one such architecture used for volumetric image segmentation termed as V-Net [23].

In the compression path, every convolution step uses a volumetric kernel of size 5x5x3 voxels. The resolution of the data is reduced by performing convolution with a kernel of size 2x2x2 voxels with a stride of 2 as shown in Figure 2.19. The size of the resulting feature map after this operation is halved as the feature is extracted by only considering nonoverlapping 2x2x2 volumes patches.

The advantage of replacing pooling layers with convolutions is that it saves memory during training.Also, there will be no switching in mapping the output of pooling layers back to their inputs during back-propagation . It can be better understood while applying transpose-convolution in place of unpooling operation. During each stage in the V-net compression path in Section 2.1.21, the number of feature channels is doubled. Non-linearities like ReLU [27] are applied through the whole network.

The decompression path of the network in Section 2.1.21, includes extracting features and expanding the spatial support of the low-resolution feature maps to gather and assemble the useful information to produce an output with two-channel volumetric segmentation. At each stage, a transpose convolution is applied to increase the size of inputs. This operation is followed by one or three convolutional layers with half the kernels compare to the previous layer. The residual function is learned similarly to the left part of the network.

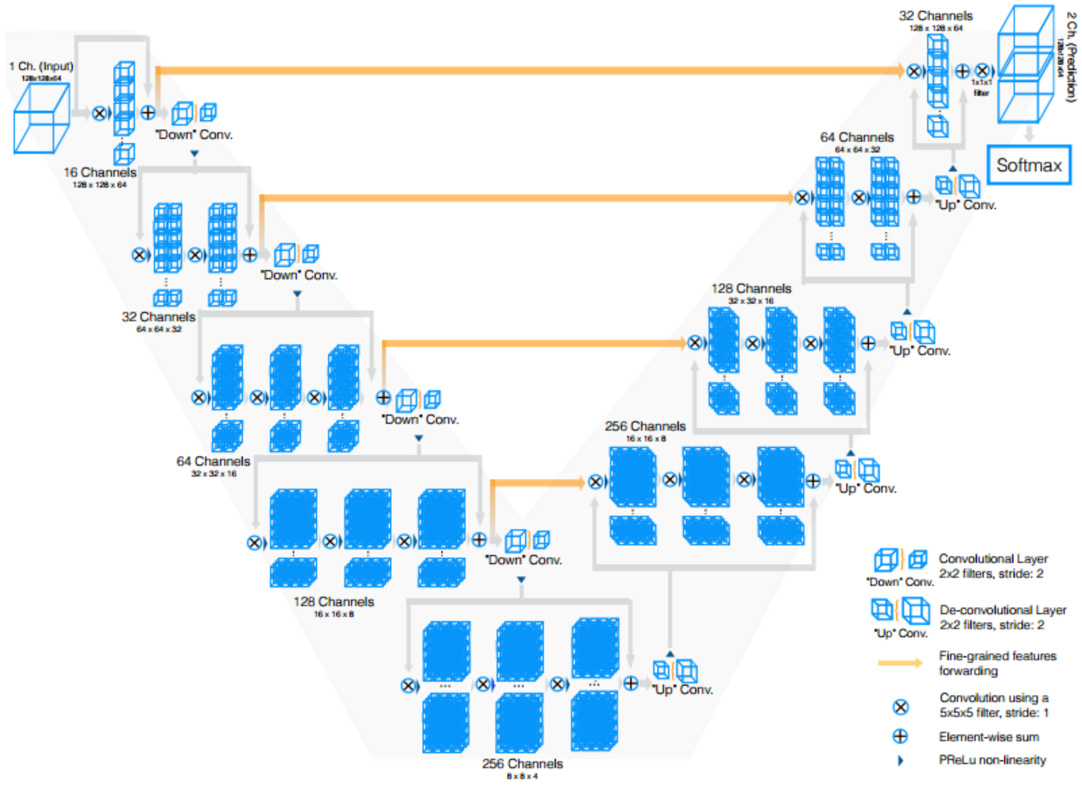


Figure 2.19: Illustration of V-Net architecture which uses 3D image data and perform volumetric convolutions. The contracting path involves 3D convolutions with applying ReLU nonlinearity and downsampling the image using max pooling. The orange arrow are used for concatenating the image border pixel information from the contracting path to the expansive path accounting for any information loss. The de-convolutions operation up-samples the image and finally softmax activation function maps the output with the desired target [23].

The last convolutional layer of the network contains a kernel size of 1x1x1, which computes the two feature maps, producing an output with the same size as input volume. The output feature maps can be transformed into segments of the foreground and background pixels by applying the soft-max activation function, with one voxel at a time. The features from the encoder path are forwarded to the decoder using the skip connections represented in the Figure 2.19 by the horizontal orange arrows. These skip connections will help in gathering fine details that can be lost in the compression path, improving the quality of the segmented mask's final prediction. These connections also help in improving the convergence time in medical image analysis, which mostly uses 3D image data [23].

Performance Metrics for image segmentation architectures

Image semantic segmentation algorithms generally face the problem of dealing with class imbalance as compare to other image analysis algorithms [8]. Class imbalance occurs where the number of background pixels are comparatively higher than the pixels in the infected area in a network dealing with images.

Usually, the performance metric chosen in such segmentation models is accuracy, which is defined as the fraction of pixels correctly classified. However, accuracy is not an ideal measure of network performance in the segmentation of images. To explain this, consider an image having 9000 pixels in the background class and 1000 pixels in the infected tissue class. In this case, a network that classify all pixels as background pixels will achieve an accuracy of 90% and ultimately failing in the task of identifying the infected area class pixels.

Due to this, we are introducing several other performance metrics for evaluating network performance, which are designed for binary classification problems. Consider a binary classification task of cancer detection in medical image analysis; pixels in an image are grouped as cancerous (positive class) and non-cancerous (negative class). It is essential to understand four terms that are a fundamental part of any performance metric, and they are *true positives*, *false positives*, *true negatives*, *false negatives*. Refer definitions 2.1.4 - 2.1.6 for a complete understanding of these terms:

Definition 2.1.4 (True positives)**.** The number of true positives (TP) is the number of pixels that belongs to the positive class that are classified correctly as a member of that class.

In our case, TP is the count of instances that were correctly classified as cancerous (positive class).

Definition 2.1.5 (True negatives)**.** The number of true negatives (TN) is the number of pixels that belongs to the negative class and are correctly classified as members of that class.

In our case, TN are the instances that were correctly classified as non - cancerous (negative class).

Definition 2.1.6 (False negatives)**.** The number of false negatives (FN) is the number of pixels that belongs to the positive class that are classified wrongly as members of the negative class .

In our case FN are the instances that are wrongly classified as non-cancerous.

Definition 2.1.7 (False positives)**.** The false positives (FP) are the number of pixels that belongs to the negative class, but they are classified wrongly as members of the positive class.

In our case, FP are the instances that were classified wrongly as cancerous (positive class) but belongs to a non-cancerous (negative class).

Sensitivity and *Specificity* are among the two very common performance metrics used in image segmentation tasks [55]. These are defined using Definition 2.1.8 and Definition 2.1.9.

Definition 2.1.8 (Sensitivity)**.** The *sensitivity* also known as *recall* is the true positive rate (TPR) of a segmentation model. It measures the proportion of positive pixels, that are correctly classified as positives by the network. Mathematically, it is represented as,

$$TPR = \frac{TP}{TP + FN} \tag{2.27}$$

here, TP is the number of true positives and FN is the number of false negatives.

Definition 2.1.9 (Specificity)**.** The *specificity* is the *true negative rate* (TNR) of a segmentation model. It measures the proportion of the negatives that are correctly classified as negatives by the network. Mathematically, it is represented as,

$$TNR = \frac{TN}{TN + FP} \tag{2.28}$$

here, TN is the number of true negatives and FP is the number of false positives.

One other performance metrics that is used to measure performance of segmentation model is termed as *precision*, or *positive predictive value* (PPV) [55], which is explained by the Definition 2.1.10.

Definition 2.1.10 (Positive predictive value)**.** The *positive predictive value* (PPV), or *precision*, is a metric, computed as the proportion of the total number of positive samples that a network predicts correctly divided by the total number of predicted positive samples.

In our case, it defines the possibility of a positively predicted pixel that belongs to the positive class (cancerous). Mathematically, it is expressed as,

$$PPV = \frac{TP}{TP + FP}, \tag{2.29}$$

here, TP is the number of true positives and FP is the number of false positives.

Dice similiarity coefficient (DSC) also known as, *overlap indices*, *Dice score*, *F-score* and F_1*-score* is a very popular metric used in segmentation tasks [56], [55].

Definition 2.1.11 (Dice score)**.** The *Dice score* (DSC) is computed as the harmonic mean of the precision (PPV) and the sensitivity (TPR). Mathematically, DSC is expressed as,

$$DSC = \frac{2}{\frac{1}{TPR} + \frac{1}{PPV}} = \frac{2TP}{2TP + FN + FP}, \tag{2.30}$$

here, TP is the number of true positives, FN is the number of false negatives and FP is the number of false positives.

The value of the *Dice score* ranges between 0 and 1 [56].

The F_β score is first introduced in [57], is also a metric used in image segmentation tasks, which is defined in Definition 2.1.12.

Definition 2.1.12 (F_β score)**.** The F_β score is calculated as the weighted harmonic mean of the precision (PPV) and the sensitivity (TPR). Mathematically, it is expressed as,

$$F_\beta = \frac{(\beta^2 + 1) \times PPV \times TPR}{(\beta^2 \times PPV) + TPR} = \frac{(1+\beta^2)TP}{(1+\beta^2)TP + \beta^2 FN + FP}, \tag{2.31}$$

here, TP is the number of true positives, FN is the number of false negatives and FP is the number of false positives.

β is the configurable parameter [57] that assigns relative weights to precision and recall. For example, if β has a low value of 0.5, it will assign more weight to PPV and less to TPR, however a high value of β such as 2.0 will assign more weight TPR and less weight to PPV.

Chapter 3

Code

3.1 The *deoxys* framework

The framework used for implementing volumetric delineation of cancer tumors of PET/CT images using V-Net is named *deoxys* and is developed by *Ngoc Huynh Bao*. The framework is build using Python version 3.7 and is based on Keras with Tensorflow 2.0.0 backend. Detailed information about the installation and usage of the framework is available on the GitHub page `https://github.com/huynhngoc`.

The framework aims to implement any model using CNN. Model creation is the first task of any segmentation procedure, here the model refers to any CNN model like sequential CNN, U-Net CNN, or customized CNN like V-Net. As we are dealing with 3D image segmentation, a V-Net, as discussed in section 2.1.21, is created, which is the first modification in the framework according to the requirement. The other change done on the original structure is the making of HDF5 (Hierarchical Data Format) Reader. The architecture with the modification is shown in the Figure 3.1.

3.1.1 Architecture Loader

The first change in the original framework is done by creating a V-Net architecture capable of handling 3D images for delineation, as shown in the Figure 3.1, section Architecture loader. The user can decide the architecture of the model by choosing the different network hyperparameters such as layers, activation and loss function, optimizers, regularisers, performance metric and callbacks, which all act as a wrapper for Keras model. The main task of the loader is to be able to create a Keras model from a configurable JSON file that contains the architecture of the model as described in Appendix A.

3.1.2 Model

This module should be a wrapper of the Keras model. Firstly, It contains all the necessary methods of a keras model such as loading a model, saving of model to a file, fitting the model with data, predicting the desired target and hence evaluating the performance of the current state of the model. Secondly, It also contains a data reader and preprocessor used if any preprocessing is needed in the data.

Data Reader

The task of the data reader is to provide input data required for training and evaluating a model. This data reader should split the input data into three sets: training, validation, and test. These datasets are wrapped into a Data generator. Since we are dealing with 3D medical image data, which is very large and causes memory issues, a Python generator is used to deal with memory related problems by feeding small parts of data to the network. This framework enables the use of large data by storing them into HDF5 format, as discussed in section 3.2. Also, it transforms the data into HDF5 form before feeding it to the data reader. The second modification implemented in the framework is by creating an HDF5 Data Reader, which processes data from an HDF5 file. The general structure and transformation of data into the HDF5 file format is described in Appendix B.

3.1.3 Single and Multiple experiments

The single experiment performs one experiment at a time. It has the capability of logging the complete performance of a given model by using the Keras callbacks, saving model at disk at a given checkpoint. Users can visualize the performance and predictions resulting from the model and also obtain the best model of each performance metric based on log files. However, in multiple experiments, several single experiments execute simultaneously.

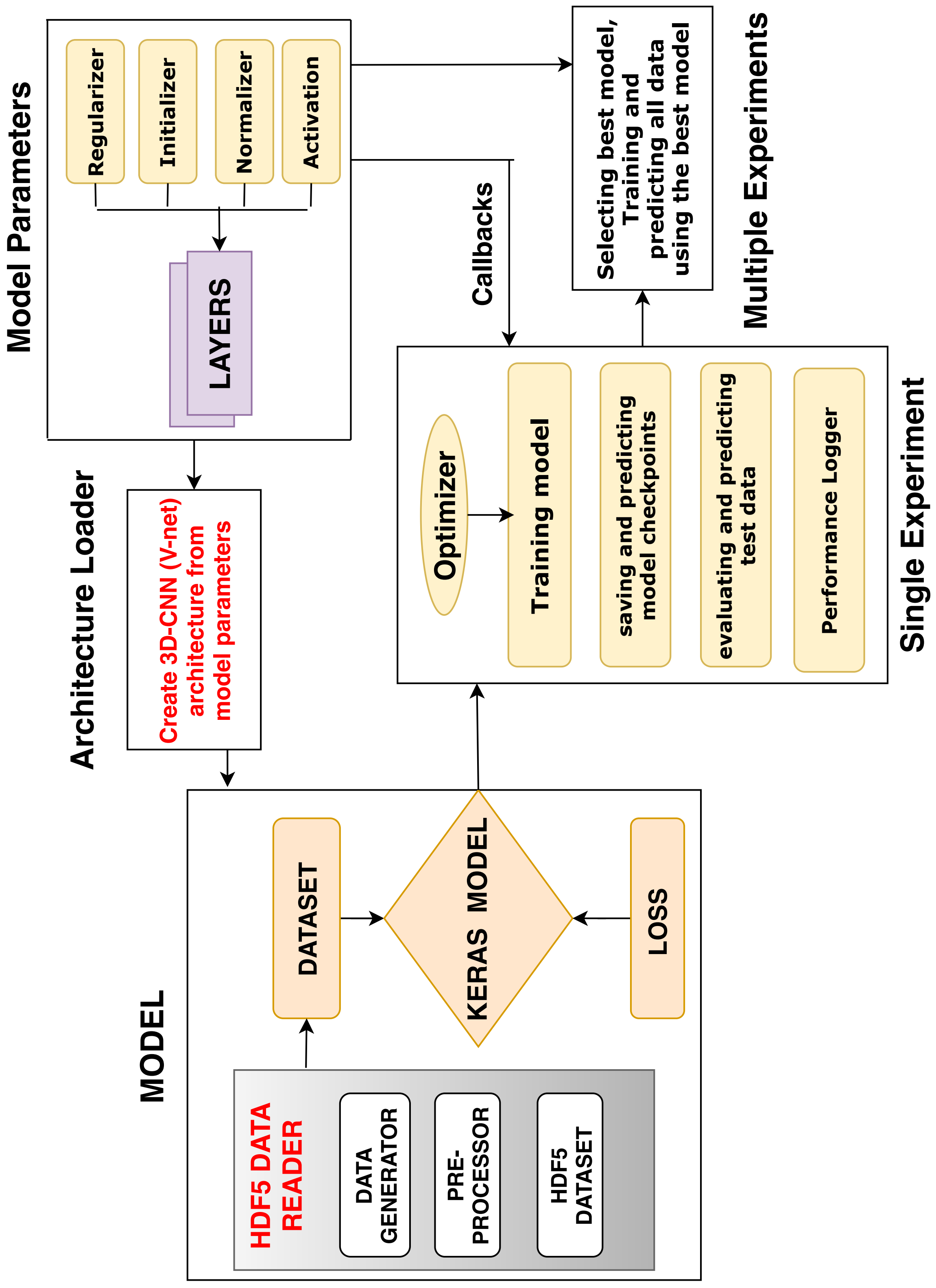


Figure 3.1: Flowchart illustrating the different modules of *The deoxys Framework* that are used to perform experiments on Head and neck cancer dataset. The different modules and their role in the delineation process is explained in section 3.1.

3.2 The HDF5 format

It is essential to keep the dataset in the project organized when dealing with deep learning problems with large datasets. In the project, we are using the HDF5 file format (or Hierarchical Data Format) [58]. For a deeper understanding of this file format and how to use it with Python, refer *Python and HDF5: Unlocking Scientific Data* by Collette [59].

The implementation of V-Net as discussed in section 2.1.21, using the *deoxys* framework needs images that should be converted into HDF5 format, which is the main requirement of using this framework.

HDF5 file format, known as Hierarchical Data Format, is a versatile file format with the capability of representing complex data objects and a wide variety of metadata [59]. It stores the data in binary form with no limit on the size of the image file. It can handle different types of data, making it very flexible and efficient for dealing with high volume and complex data.

One main component of HDF5 files is *datasets*, which are data arrays similar to NumPy arrays that can be stored on disk directly or in compressed form. Different methods of compression are available in the `h5py` library like chunked compression. The syntax for slicing and extracting elements at a particular index can be easily used with datasets similar to that performed on NumPy arrays, removing the need for reading and writing a complete array every time from disk. An example of creating a dataset in `h5py` file is described in Example 3.2.1

Example 3.2.1 (HDF5 datasets)**.**

```
import h5py
import numpy as np

arr=np.ones((5,2,3))
with h5py.File('h5py_test.h5','w') as hf:
    hf.create_dataset('dataset',data = arr)


 with  h5py.File("h5py_test.h5", "r") as hf:
    array_slice = hf["dataset"][0,:2,:1]
```

Here we can see that we have created a `h5py.File("h5py_test.h5", "w")` file which is being opened using the append mode (represented by 'w') and created a dataset inside this file with data from a multidimensional array. We can read the contents of this dataset from `h5py.File("h5py_test.h5", "r")` in read mode (represented by 'r').

Other main component of any HDF5 file are *Groups* as described in Example 3.2.2 ,which are containers similar to folders in any file system. They can store others groups and datasets. These groups are responsible for building up the hierarchical structure of any HDF5 file, creating objects that are nicely organized as groups and subgroups.

Example 3.2.2 (HDF5 Groups)**.**

```python
import h5py
import numpy as np

arr1= np.ones((5,3,2))
arr2 = np.ones((4,2,3))

#create groups
with h5py.File('test_h5py.h5','w') as hf:
    group = hf.create_group('group')
    data_array1 = group.create_dataset('data_array1', data =
       arr1)
    data_array2 = group.create_dataset('data_array2',data = arr2
       )

    data_array1[...] = arr1
    data_array2[...] = arr2

with h5py.File('test_h5py.h5','r') as hf:
     array_slice = hf['group/data_array1'][0,:2,:]
```

In the above example we created a group in `h5py.File("h5py_test.h5", "w")` and then created two datasets within that group with data from the multidimensional arrays arr1 and arr2. Then we read this data from `h5py.File("h5py_test.h5", "r")`. Data at any index can be read using slicing similar to Numpy arrays.

Chapter 4

Experimental Set-up

4.1 Experiments

4.1.1 The Dataset

The data provided for implementing the *deoxys* framework contains 3D CT/PET images and segmentation masks made by an expert of 197 patients that have gone through treatment at the Oslo university Hospital, the Radium Hospital. The goal of this project is to delineate affected lymph nodes and gross tumor volume (GTV). The model used the union of the ground truth and segmentation masks where multiple delineations existed. Each image was cropped in 3D to reduce the class imbalance. The final 3D images contained between 0.02% and 32% tumor/lymphatic node voxels.

Data has initially been in Matlab Format. Before proceeding with any modification in data, it is converted into an HDF5 format. The images consist of two channels CT and PET. These images are stacked over each other such that 3D image dimensions after stacking become: width x height x depth x channel, where Channel = 2 (CT and PET). Users should not confuse with RGB (Red, Blue, Green) channels of color images with these channels provided, and they are the different values of the image. The targets (segmentation masks) were formed by the union of the tumors and lymphatic nodes for each patient, if multiple delineations are present in the image. The one thing to keep in mind while creating different groups for an HDF5 file is that images and targets should have the same dimension.

Table 4.1: Total number of patients in each dataset used in the network.

Dataset	No.of patients
training	142
validation	15
testing	40

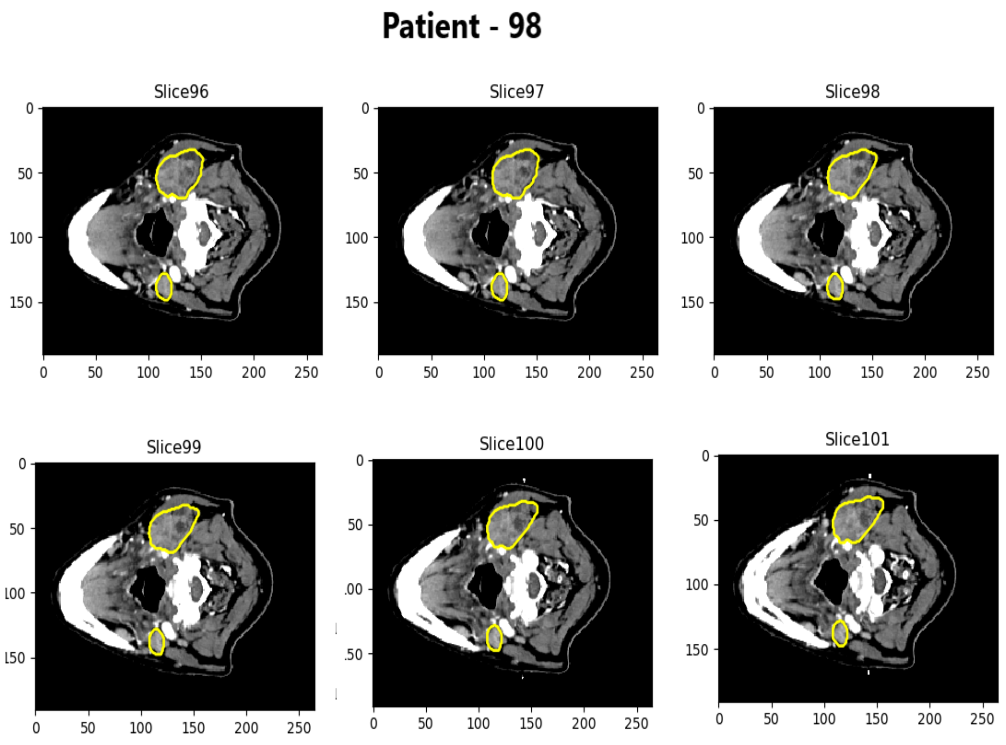
In the experiment, we created three datasets in which the data is divided, which is a need for any deep-learning task. The first dataset is the training dataset used to train the network, and the second is a validation set used for parameter selection, avoiding overfitting, and tuning the parameters of the model, also used for comparing models. The final dataset is the test set, use to assess the equality of the final model. A summary of the different datasets and their sizes is given in the table 4.1.

The delineation process uses the two-class classification for the dataset, one is healthy tissue, and the other is affected tissue. Any voxel in the head and neck images belongs to the affected tissue class if it is delineated as either tumorous or lymph nodes. The structure of each dataset file used in the project is as follows: Three groups in the file root ”training” contain 142 patients, ”test” that comprises of 40 patients and a ”validation” that includes 15 patients, without stratification based on tumor stages. Some image slices of patients 98 and 229 from validation set is described in Figure 4.1 and Figure 4.2. Each group here contains three dataset information: images, masks, and patient id’s. The summary of the structure of groups and datasets used is described in the table 4.2.

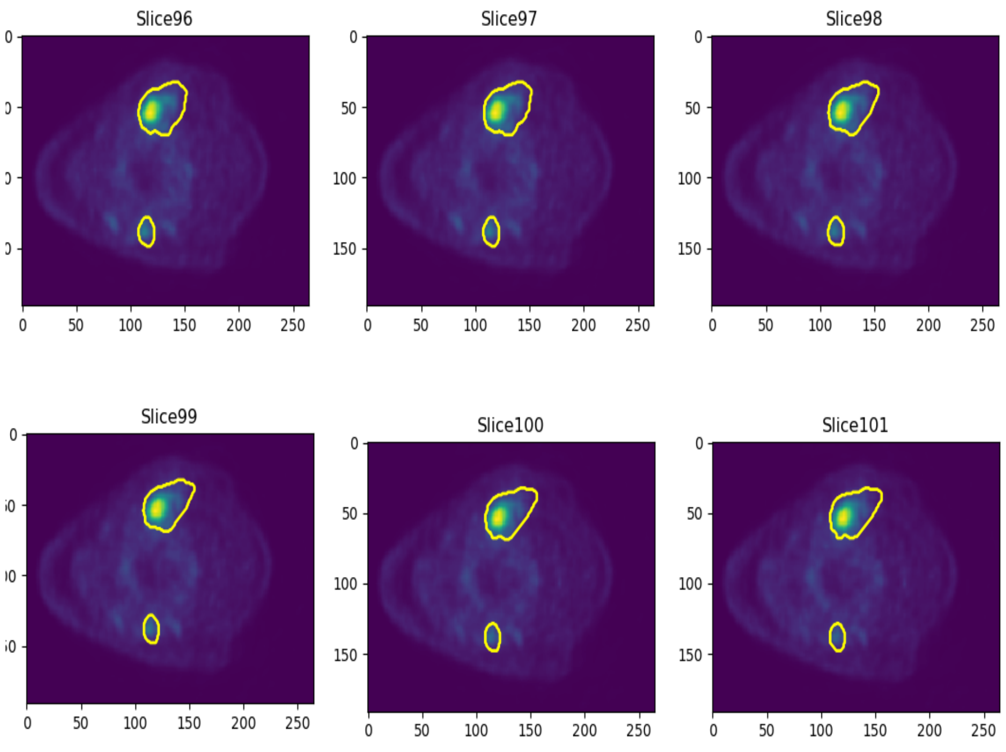
Table 4.2: Demonstrating file structure of one group or fold used in the experiment.

Dataset	Shape	Datatype	Contents
images	`[n_patients, x, y, z, c]`	float32	The input images.
masks	`[n_patients, x, y, z, m]`	float32	The segmentation masks.
patient_id	`[n_patients]`	uint16	The patient ID number

Table 4.2 describes the complete overview of the structure of the HDF5 files used in the experiments. `n_images` are the total number of images in the particular dataset, `x` is the number of pixels in the x direction, `y` is the number of pixels in the y direction, `z` is the number of pixels in the z direction, We have three directions of pixels as the experiment is dealing with 3D images. `c` is the number of input channels used (for PET/CT this is 2) and `m` is the number of segmentation masks.

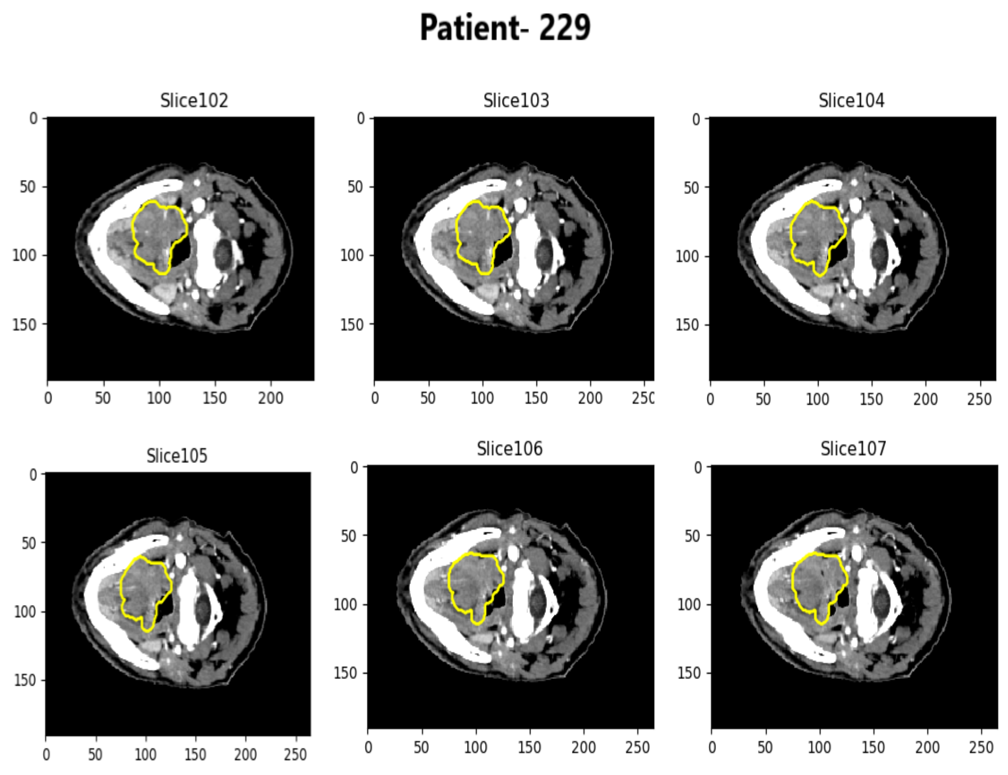


(a)

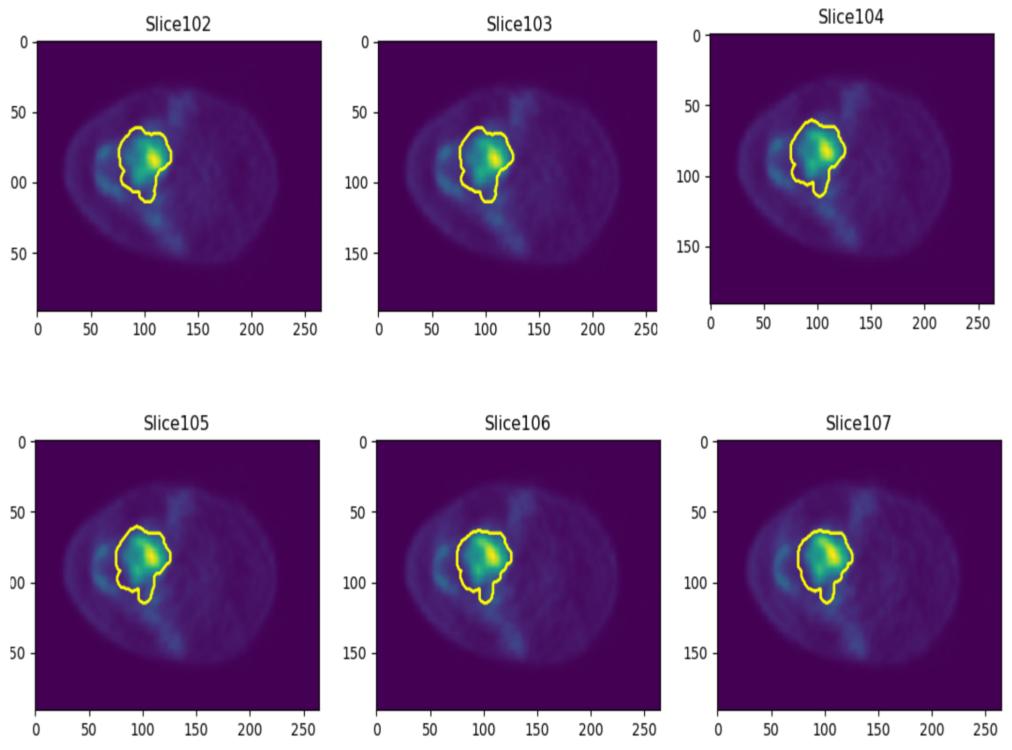


(b)

Figure 4.1: Demonstration of image slices showing (a) CT slices and (b) PET slices of head and neck cancer patient 98 of the validation set describing the gross tumor volume marked with yellow color.



(a)



(b)

Figure 4.2: Demonstration of image slices showing (a) CT slices and (b) PET slices of head and neck cancer patient 229 of the validation set describing the gross tumor volume marked with yellow color.

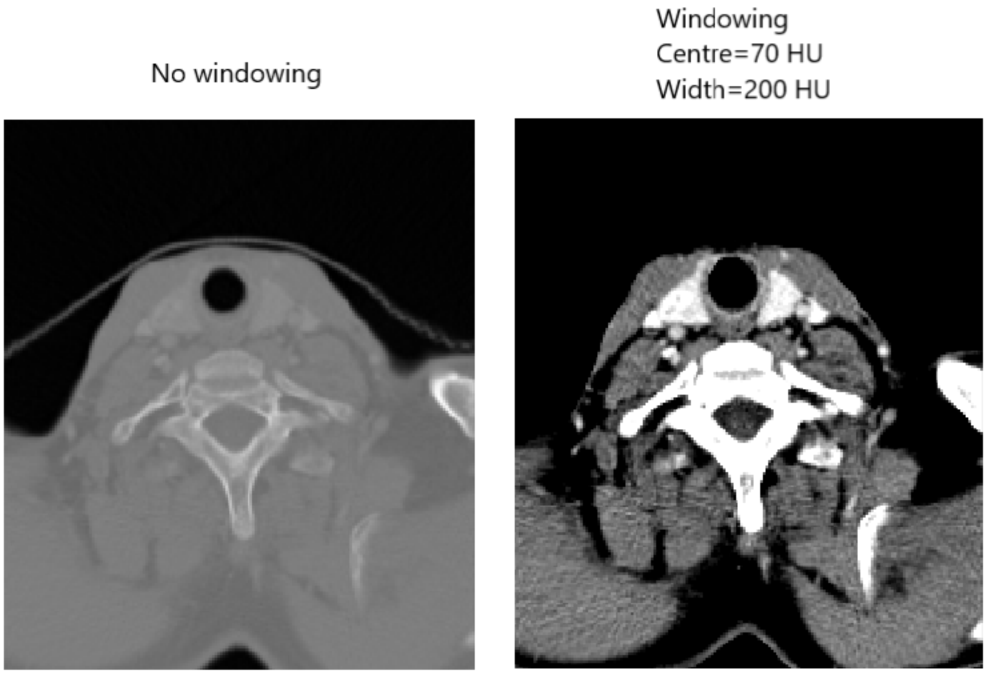


Figure 4.3: Illustration of Hounsfield windowing preproccessing on head and neck CT image. The left image represents a CT image slice with full dynamic range, whereas the right image shows the same CT image slice but with a reduced dynamic range.

4.1.2 Preprocessing

Two types of preprocessing was performed on the 3D head and neck cancer dataset in our experiments. Firstly, all the 3D PET and CT images have been resampled to isotropic voxels of size 1x1x1 mm^3. Also, PET image values are modified to standardized uptake value (SUV) with normalized body weight by an expert [60].

Secondly, Thresholding was used for reducing the dynamic range of the CT channel known as Hounsfield windowing. Possible combinations of preprocessing (i.e., PET/CT, PET/CT + windowing) were performed on the 3D image dataset in the experiments. The parameters of Hounsfield windowing like window center and window width used in the experiments were set after consulting with a radiologist. The window size was set to surround most of the soft tissue dynamic range, and the window centers were chosen nearly equal to the average and median tumor value. The parameters used for windowing are illustrated in the Table 4.4. An illustration of Hounsfield windowing applied on a head and neck cancer patient is shown in the Figure 4.3.

4.1.3 Model Parameters

Architecture

Several experiments were run using the V-Net architecture [23] , as discussed in the Section 2.1.21 with the same number of layers and convolution per layer. The number of filters chosen respectively for each downsampling and upsampling layer for the architecture are [32, 64, 128, 256, 512]. The architecture summary is described in the table 4.5. All the hyper-parameters used during experiments are listed in the table 4.4.

4.1.4 Layer Type

The layers used for performing experiments are termed as convolutional layers as discussed in section 2.1.12. These layers consist of kernels of size 3 x 3 x 3 which are followed by a ReLU nonlinearity as discussed in section 2.1.3 and finally batch normalization as discussed in section 2.1.15.

4.1.5 Loss Function

Two different type of loss functions are used while performing experiments. The detailed information about these losses is discussed in section 2.1.5. The loss function used for the experiments is the Dice loss as stated in section 2.1.11 , the F_β with with $\beta = 2$.

4.1.6 Training Procedure

Initial experiments are performed on a local machine with processor - Intel(R) Core™ i5-8250U CPU and 8 GB RAM. Operations started with a 4-layer V-net architecture [23], with only one filter in the first convolutional layer. Then we moved to 5-layer V-Net architecture with one filter in the first layer. The summary of the 5-layer V-Net architecture used is described in the table 4.3. Since our dataset contains 3D images, running experiments with a higher number of filters and layers required more memory and hence needed a system with higher computational power. Still, we tried to increase the number of filters in the first layer to

Table 4.3: Overview of the V-Net architecture used in running experiment on local machine.

Name	Type	Input	No. output channels
Conv 1	Convolutional	Input image	1
MaxPool 1	Max Pooling	Conv 1	1
Conv 2	Convolutional	MaxPool 1	1
MaxPool 2	Max Pooling	Conv 2	1
Conv 3	Convolutional	MaxPool 2	2
MaxPool 3	Max Pooling	Conv 3	2
Conv 4	Convolutional	MaxPool 3	2
UpConv 1	Upconvolutional	Conv 4	2
UpConv 2	Convolutional	UpConv 1, Conv 3	2
Upconv 3	Convolutional	Upconv 2 ,Conv 2	1
Conv 5	Convolutional	Conv 1, Upconv 3	1

The Convolutional and upconvolutional layers are standard 3D convolutional layers.

4. The operation leads to memory-related issues, which signifies that computation of such a V-net model for delineating 3D images needs a system with high memory capacity and fast processing speed.

The experiments are then performed using the Orion computer cluster, an open-source platform hosted by NMBU (Norwegian University of Life Sciences) and operated by CIGENE. The Orion cluster is a remote server that helps a user to run experiments with large CPU memory and GPU's for fast processing speed. It is a Linux operating system that needs SSH (Secure Shell) command to build a secure encrypted connection with the remote Orion cluster. Firstly, a user should establish a remote connection using Visual studio code and entering the credential provided by the Orion team. As soon as a remote connection builds, users get access to a home directory where one can edit scripts, manage different files, and submit different jobs to the cluster environment. The next step is to have all the source code, the head and neck dataset, and JSON configuration files of our experiment set up in the Visual studio code application. Every single experiment a single GPU.

This information is defined in the slurm script before submitting the jobs. The

Table 4.4: Overview of the hyperparameters used for the V-Net architecture.

Hyperparameter	Value(s)
Learning rate	[0.0001, 0.00001]
Optimiser	Adam
Nonlinearity	ReLU
Normalizer	Batch Normalisation
Initializer	Normally distributed He
Dropout 3D regularizer	Dropout rate: 0.3 and 0.5
Layer type	Convolutional layers
Window center	70 HU
Window width	200 HU
Loss	$[F_1, F_2, F_4]$
Batch size	2
Number of epochs	$100 - 300$
Iterations between checkpoints	10

slurm script, also known as the job scheduler application, is used to get exclusive access to the cluster resources. A user should also create a Singularity container that contains all the necessary libraries, software, scientific workflows needed while running our experiment. The *deoxys* framework, as discussed in the section 3.1, is also installed using a Singularity container. Finally, a slurm script is submitted to the Orion cluster to run the experiments on a server machine. Sample slurm script and singularity container are described in the Appendix C. The complete code used for performing experiments, all the parameters including JSON configuration files, slurm script, and singularity container, can be found on the GitHub page: (https://github.com/afreenmirza3010/msc-cluster-code/). Different Orion-related queries and steps can be found on the Orion cluster GitLab page: (https://gitlab.com/cigene/computational/orion-support/-/wikis/home). [1]

[1]Visual studio code is a code editor software that is available for free, with several features like debugging, code compiling , code refactoring, and embedded git. Also, it has a remote extension that helps users to open a remote folder on any remote machine. In our experiments, the remote device is the Orion cluster.

Table 4.5: Overview of the V-Net architecture used in the project.

Name	Type	Input	No. output channels
Conv 1	Convolutional	Input image	32
Conv 2	Convolutional	Conv 1	32
MaxPool 1	Max Pooling	Conv 2	32
Conv 3	Convolutional	MaxPool 1	64
Conv 4	Convolutional	Conv 3	64
MaxPool 2	Max Pooling	Conv 4	64
Conv 5	Convolutional	MaxPool 2	128
Conv 6	Convolutional	Conv 5	128
MaxPool 3	Max Pooling	Conv 6	128
Conv 7	Convolutional	MaxPool 3	256
Conv 8	Convolutional	Conv 7	256
MaxPool 4	Max Pooling	Conv 8	256
Conv 9	Convolutional	MaxPool 4	512
Conv 10	Convolutional	Conv 9	512
UpConv 1	Upconvolutional	Conv10	256
Conv 11	Convolutional	UpConv 1, Conv 8	256
Conv 12	Convolutional	Conv 11	256
UpConv 2	Upconvolutional	Conv12	128
Conv 13	Convolutional	UpConv 2, Conv 6	128
Conv 14	Convolutional	Conv 13	128
UpConv 3	Upconvolutional	Conv14	64
Conv 15	Convolutional	UpConv 3, Conv 4	64
Conv 16	Convolutional	Conv 15	64
UpConv 4	Upconvolutional	Conv14	32
Conv 17	Convolutional	UpConv 4, Conv 2	32
Conv 18	Convolutional	Conv 17	32
Conv 19	Convolutional	Conv 18	1

The Convolutional and upconvolutional layers are standard 3D convolutional layers.

4.1.7 Model Performance analysis

The average Dice score value as described in section 2.1.11 is used to analyze and compare performance between different models. The model training is done using the most suitable hyperparameters combination, and the dice score on the validation set is recorded. The best model is then evaluated based on model performance on the validation set. The model with highest average Dice is used to run the final experiment on the test data to get the model's performance on unseen data.

4.2 Tensorboard Profiling Outcomes

Deep learning algorithms are very memory-intensive, especially when we are dealing with 3D images. Thus, it is essential to quantify the performance of the deep learning algorithm to ensure that we are running the most optimized version of the model. Hence we need a system to track Keras model performance. For this purpose, Tensorflow Profiler is used, which is embedded within Tensorboard. Profiling the model performance can be understood as a tool that helps a user understand the hardware resource consumption, i.e., the time and memory of the various operations in the model. Profiling allows users to find the bottlenecks in the model that produces delays. Hence, a user can resolve the bottlenecks and make the model works faster.

In our experiments, we used Tensorboard profiling to understand the bottlenecks that are causing delays in the performance. Generally, In any CNN model, it has been found that the max pooling operations took more time during the back-propagation step, hence causing delays. As suggested by Milletari *et al.*, the solution for this is to replace max-pooling layers with convolution layers that can make the computation faster with smaller memory footprint during training [23]. There will be no mapping of the pooling layer's output back to their input as needed for backpropagation.

Chapter 5

Results

5.1 Initial Experiments

Initial experiments on the 3D head and neck datasets were performed on a local machine using a 5-layer V-net architecture as described in section 4.1.6 . The performance is measured using the Dice score as discussed in section 2.1.11 using the F_β values of each trained model. Summary statistics like training Dice score, training loss, validation Dice score, and validation binary F_β loss are recorded and stored in the log files for evaluating the performance of a given model with all the hyperparameters provided.

The plot obtained for the Dice score of the 5-layer V-net model ran for 30 epochs is described below Figure 5.1. We were only able to run 30 epochs with a model described in Table 4.3 due to memory issues. The Dice score obtained is not satisfactory. Still, these values were increasing with epochs, which led us to run more experiments on the same dataset on a machine with high computational power and memory available.

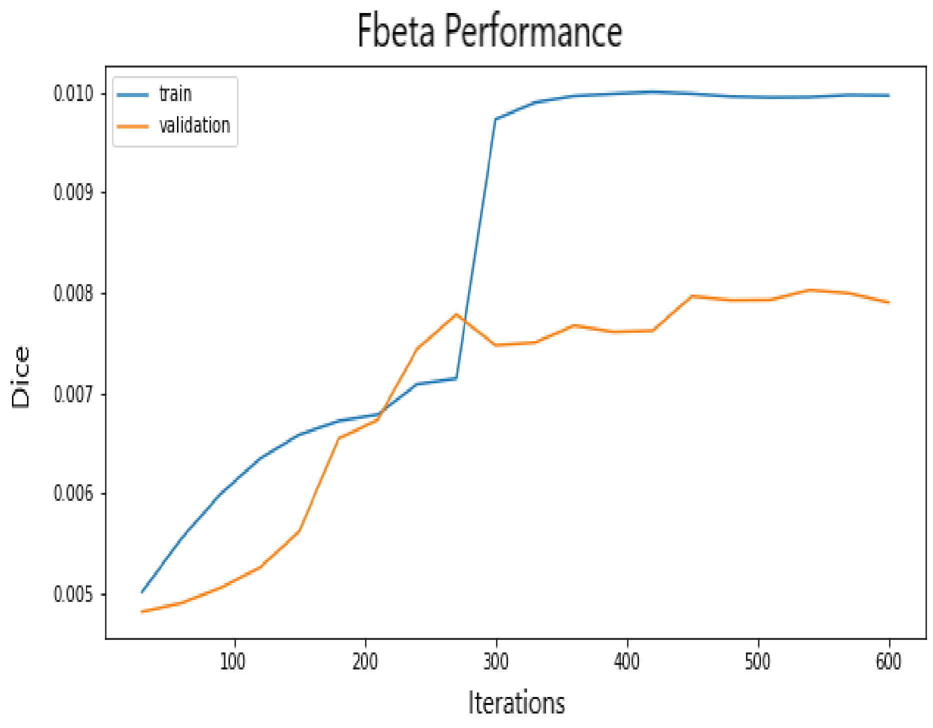


Figure 5.1: Illustration of F_β performance based on Dice score on training and validation set. The blue line shows the Dice score for training dataset. The orange line shows the Dice score for validation dataset. The model ran for 600 iterations (30 epochs). From the curve it is clear that both training and validation Dice score are increasing with iterations.

5.2 Experiments on Orion cluster

Various experiments were performed on the 3D head and neck data set using a V-Net architecture described in section 4.1.6 using the Orion cluster.

The first thing that we started experimenting with our model was with the number of filters. The first set of filters for each upsampling and downsampling convolutional layers used in the experiment was [16, 32, 64, 128, 256], respectively. Then we updated the number of filters and used [32, 64, 128, 256, 512] in the final experiment. Increasing the number of filters also increased the training time for the model. Still, it has given a adequate boost in the model performance. The final architecture summary is described in the table 4.5.

5.2.1 Model Performance on the validation set

The F_β loss and Dice score curves obtain as the result of our delineation method using [32, 64, 128, 256, 512] filters in the upsampling and downsampling layers are shown in Figure 5.2. All the experiments used Adam optimizer and a learning rate of 0.0001.

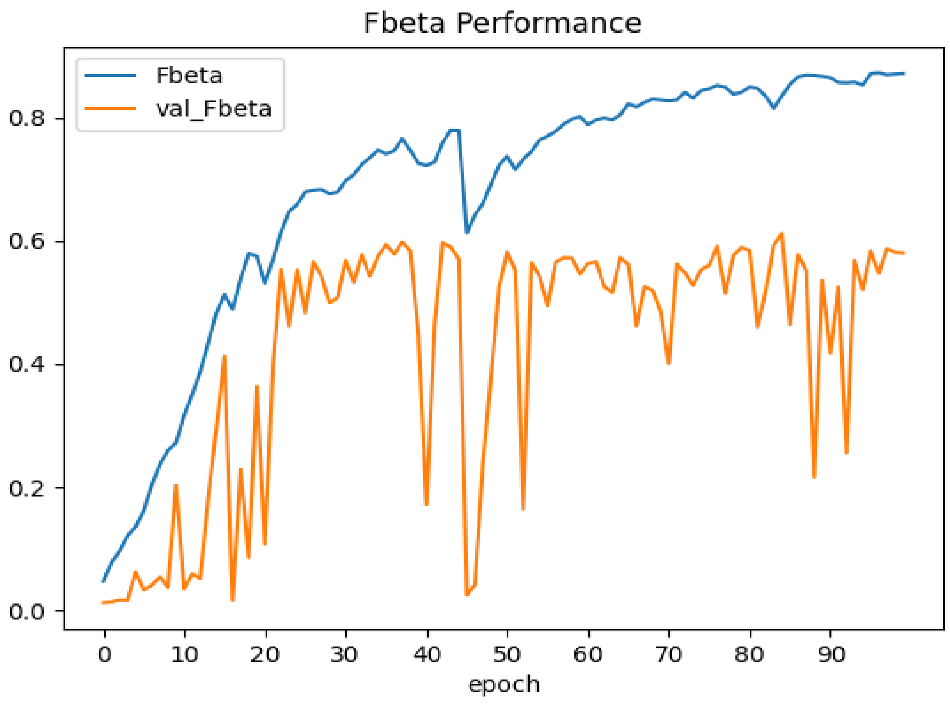
A total of 100 epochs were run on the Orion cluster to obtain the results. The model performance is measured, and all the checkpoint weights are tested and stored in logs files that are in CSV(Comma-Separated Value) format. Model checkpoints are added after every 10 epochs to capture the details of all the parameters used by the model. The highest performing model was at epoch 98, obtained from the log files. A combination of PET/CT channels is considered in all the experiments.

We also ran experiments on the dataset by increasing the number of filters in the first layer. We used [64, 128, 256, 512, 1024] set of filters in the downsampling and upsampling layers of the V-Net architecture. The summary statistics obtained is described in Table 5.2.

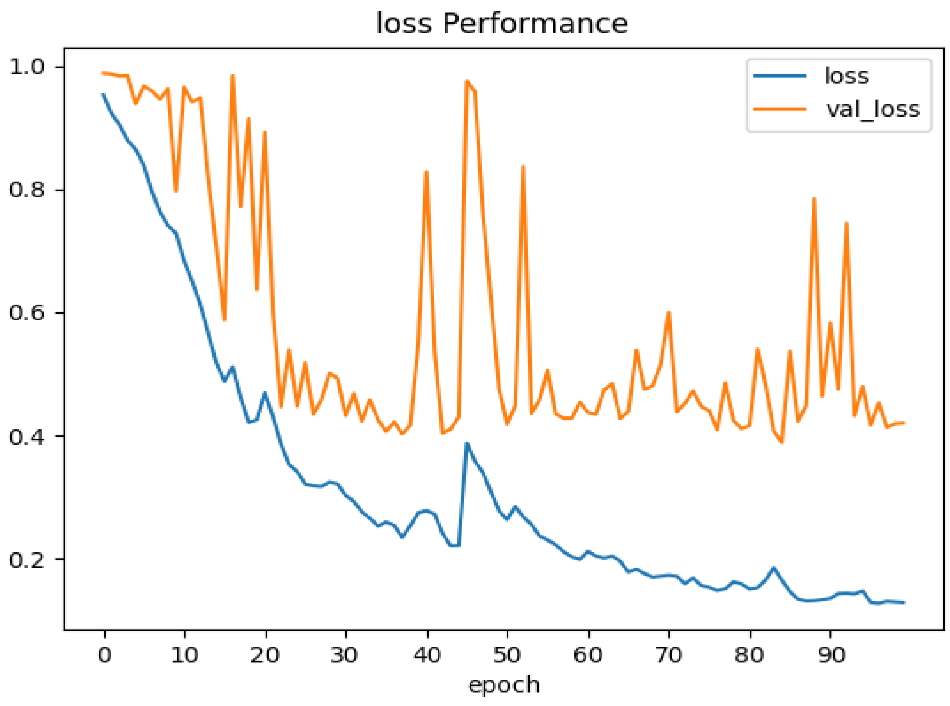
Table 5.1 describes the summary statistics of the effects of the “loss” hyperparameter on model performance. The F_β loss for all values of beta are calculated and F_2 had higher performance than Dice with respect to all performance summary statistics.

As suggested by Erden *et al.* [61], Dropout regularization as discussed in section 2.1.18 seems to enhance performance of CNN models. Hence, we applied dropout with probability 0.3 and 0.5 to the V-Net architecture. The statistics summary obtained from the experiments is discussed in Table 5.3 and 5.4. The dropout probability rate 0.5 did not perform well on our dataset.

The details of all the hyperparameters used in the V-net architecture that provided the best overall performance on the validation set (15 patients) in our experiment are described in the Table 5.6. The highest performing model was found at epoch 98.



(a)



(b)

Figure 5.2: (a)Dice curves and (b) The loss curves for a typical model. Note that the model has not converged, as the loss is still decreasing, however, the validation lines have plateaued. In both figures, the dark blue lines represents the Dice and loss performance evaluated on the training set and the orange lines show the the Dice and loss performance evaluated on the validation set.

Table 5.1: Dice results on the validation set for the “loss” hyperparameter for 32 filters in the first layer.

Loss	Value
F_2	0.6221
Dice	0.6115

Table 5.2: Dice results on the validation set for the “loss” hyperparameter for 64 filters in the first layer.

Loss	Value
F_2	0.6339
Dice	0.5704

Table 5.3: Dice results on the validation set obtained for the dropout probability rate 0.3.

Loss	Value
F_2	0.4577
Dice	0.4352

Table 5.4: Dice results on the validation set obtained for the dropout probability rate 0.5.

Loss	Value
F_2	0.0187
Dice	0.0244

Table 5.5: Dice results on the validation set for the “windowing” hyperparameter.

	Modality
Hyperparameter	PET/CT
60 HU	0.5818
70 HU	0.6136

Table 5.6: The hyperparameters of the models that achieved highest Dice score on the validation dataset.

	Modality
Hyperparameter	PET/CT
Loss	F_2
Optimiser	Adam
Learning rate	0.0001
F_2	0.6221

5.2.2 Model performance on the test set

The best predictive model is selected based on the performance on the validation set (15 patients). The network which is well optimized is then used to evaluate the test set.

The model with the highest Dice score value as mentioned in the Table 5.6 is used to run experiment on the test set (40 Patients). The summary statistics on the performance of test set is described in the Table 5.7. Slices showing the segmentation masks predicted by the PET/CT model for patient 8 are shown in Figure 5.3.

Table 5.7: Dice performance in the test set for the best models using multi-modality images

Loss	Value
F_2	0. 6750
Dice	0.6286

From the summary statistics obtained from the model with layers [64, 128, 256, 512, 1024] in the downsampling and upsampling layer as described in Table 5.2, The F_2 score measured is 0.6339, which is higher than the model with 32 filters in the first layer. However, the performance of this model on the test set scored a F_2 : 0.6010, which is less as compared to the model with layers [32, 64, 128, 256, 512]. Therefore, we proceeded with the model as described in section 4.5 for all other experiments.

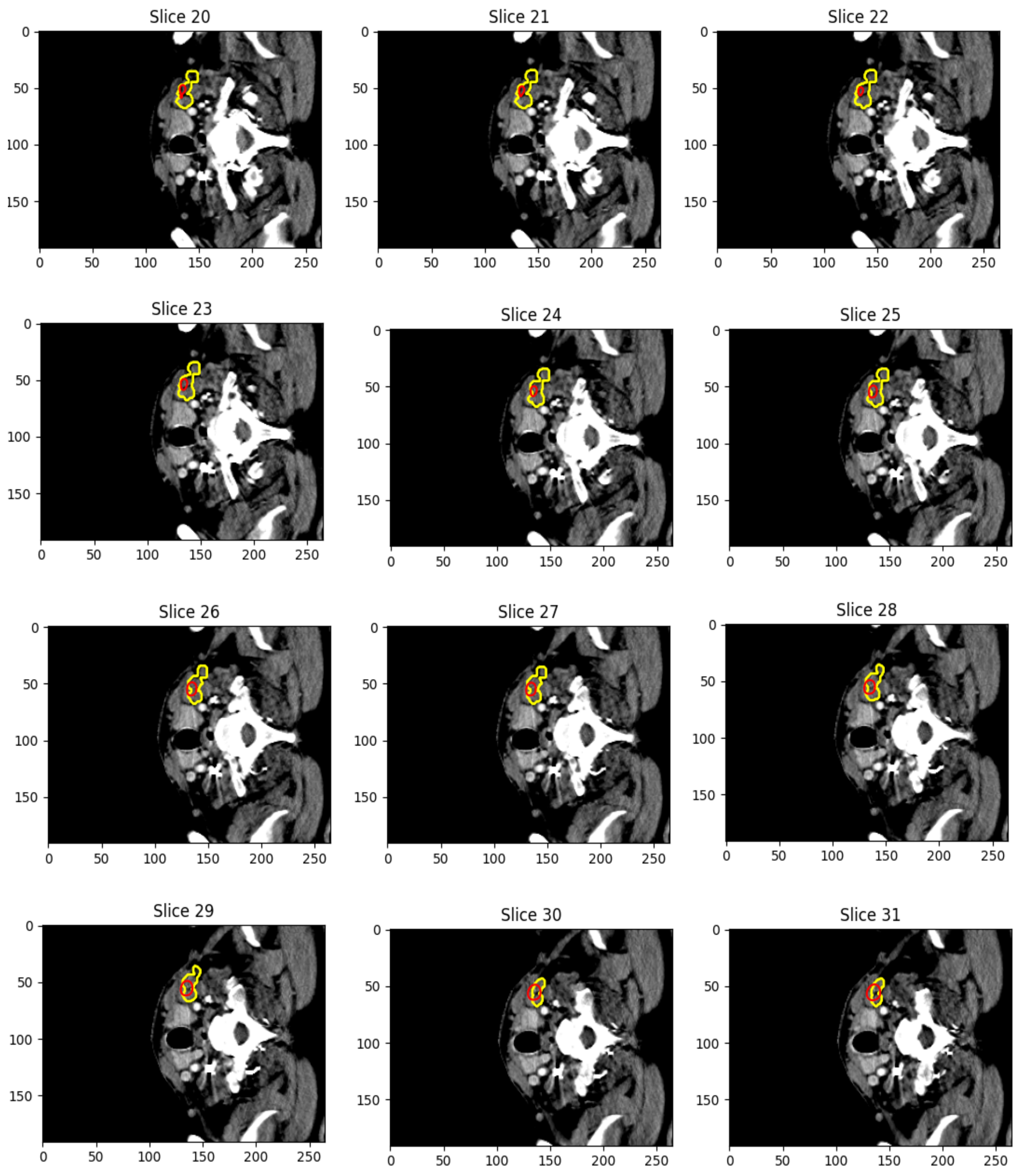


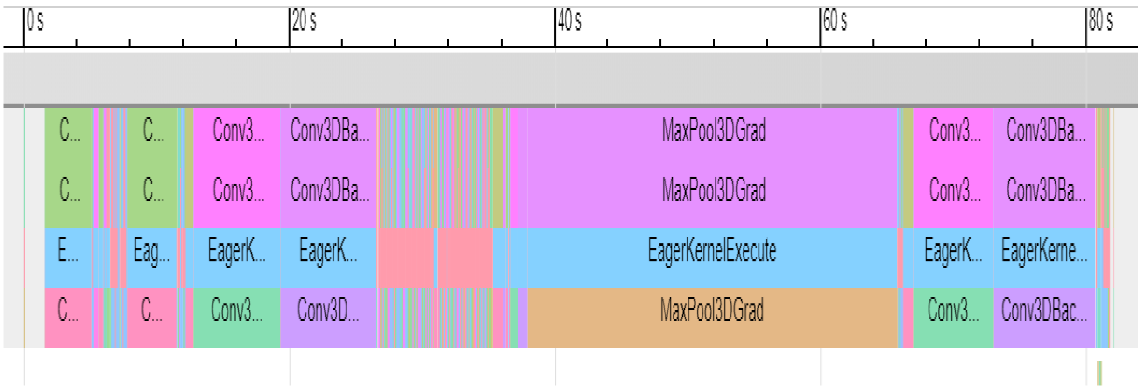
Figure 5.3: Slices showing the segmentation masks predicted by the PET/CT model for patient 8. The ground truth is represented by yellow colour and the predicted mask is represented by red colour. The average Dice for this patient was 0.435.

5.3 Tensorboard Profiling Outcomes

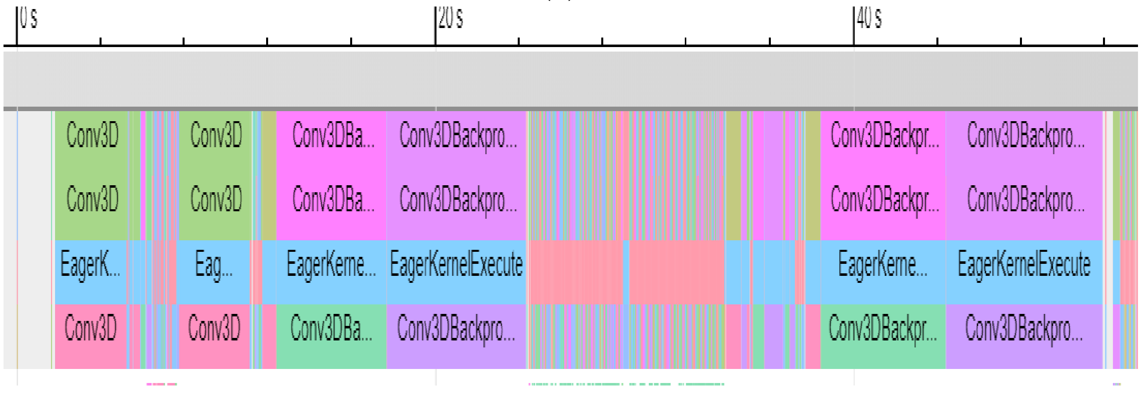
As discussed in section 4.2, we used Tensorboard profiling to find the bottlenecks in the V-net architecture, which are causing delays and consuming more memory in the overall performance of the experiments. From the Figure 5.4 (a), it is apparent that the max-pooling operation was causing delays to the model performance during the backpropagation operation and hence is the main bottleneck causing memory issues.

Hence, as suggested by Milletari *et al.* [23], we replaced the max-pooling layers with the convolutional layers and the profiler output after replacement of layers is shown in Figure 5.4 (b). It significantly reduced the overall time consumption of the experiment and decreased the memory issues in our experiments.

The performance summary of our model can we viewed using the Tensorboard profile tab. More information about using Tensorboard with Keras can be found on https://www.tensorflow.org/tensorboard/tensorboard_profiling_keras



(a)



(b)

Figure 5.4: The figures explain the performance of the V-Net model capture using the Tensorboard profiler. The figure (a) illustrates the model performance using max-pooling layers. The time taken for each step mentioned at the top of the figure. The figure (b) illustrates that the replacement of max pooling with convolutional layers significantly reduced the performance delays and, hence, the memory consumption, thus increasing overall model performance.

Chapter 6

Discussion

In medical image analysis, volumetric image data is in abundance [62]. However, annotating large volume images with segmentation labels in a slice-by-slice manner is a tedious process, as neighboring slices contains similar information. In such scenario, considering the complete image volume can provide more spatial feature information than 2D image slices.

3D neural networks enable users to extract features in an image in three dimensions and help in establishing a relationship between three dimensions (width, height, and depth). Many studies have proven that 3D neural networks have improved performance compared to 2D networks for the segmentation of tumors [23], [62].

The study already performed by Yngve Mardal Moe [48] on the head and neck data using the 2D U-net has given very good performance in segmenting tumors. It gave us great interest in extending our analysis for understanding the effect of using the 3D convolutional neural network on the same dataset considering the whole volumetric image and not slice by slice as used in the 2D network , and can compare the overall performance of the 3D model to the 2D U-net.

6.0.1 Architecture recommendation

As our project deals with 3D head and neck PET/CT images, 3D convolutional neural networks like V-Net [23], 3D Dense net [63], and AnatomyNet [64] can become a desirable choice for segmentation tumors.

However, we chose the V-net architecture in our experiments because it was easy to implement, which helped in reducing the development phase of the project. It is very similar to the 2D U-net, and can be created by replacing 2D operations with 3D operations, such as 3D convolutional layers, 3D max-pooling, and 3D up-convolutional layers and take 3D images as input.

There are other architecture suitable for 3D images that can yield better results.

An example of such architecture is the 3D Dense-Net [63], which has used a dataset containing 250 head and neck cancer patients PET/CT images. For comparison, all the configurations of Dense-Net and V-Net are kept similar in their experiments, such as filter size and strides. This architecture consists of dense blocks as well as transition down and transitions up modules. The Dice score obtain for their 3D dense network is 0.73 on the test set (75 patients) and for the 3D U-net they have a Dice score of 0.71. The binary F_β score obtained by our V-net model is 0.6750 on the test set (40 patients). Hence, 3D Dense network can be a preferable choice for our experiments for boosting the performance.

This study found that the segmentation accuracy of the dense model is better as compare to the 3D U-net. Some reasons for the better performance of the Dense net can be understood from their analysis as stated below.

Firstly, the main point in any semantic segmentation is the memory issues that are caused while increasing the network depth as the 3D convolutions increase the computational burden as we observed while performing experiments with V-Net. Memory issues are dealt using residual network with skip connections in their 3D dense network, as discussed in section 2.1.17.

Secondly, the dense block in the architecture uses extreme connecting patterns between layers that map the output of one layer to all subsequent layers by skip connections. For more information on model design, refer the Dense architecture proposed by Guo *et al.* [63]. This type of pattern connection helps each layer to gather more information from other layers hence reduces vanishing gradient problems, strengthens feature reuse and feature propagation, and thus reduces the number of trainable parameters. Therefore, this architecture can be an excellent alternative for our experiments in the future dealing with 3D images.

Another architecture proposed by Zhu *et al.* [64] is the AnatomyNet, which is commonly used for 3D image segmentation. The dataset used in the experiments contain 261 head and neck cancer patients CT images.

The AnatomyNet architecture is similar to 3D U-net except it has only one downsampling layer, which accounts for the trade-off between network learning ability and extreme GPU memory consumption. This downsampling layer is used in the first encoding block so that the subsequent blocks occupy less GPU memory. Also, the standard convolutional layers will be replaced by the 3D SE (Squeeze and excitation) [65] residual blocks known to be beneficial in learning features. This method should be tested for future work. The Dice score on the test set (10 Patients) obtained in their experiments is 0.768.

.

6.0.2 Loss Functions recommendations

The choice of loss functions is crucial in semantic image segmentation due to the need to segment small volumetric objects. The main challenge is in any segmentation process is the imbalanced data distribution, as it requires pixel-wise labeling. In our experiment, we have chosen Dice loss as the loss function as it is commonly used when the data is imbalanced.

However, Zhu *et al.* proposed a loss function that has shown better performance when dealing with 3D volumetric data, i.e., the hybrid loss [64]. Hybrid loss is a combination of Dice loss and focal loss. Here, Dice loss learns the class distribution by reducing the imbalanced dataset problem, whereas focal loss makes the model classify the poorly classified voxels better.

The Dice score on the test set (10 Patients) obtained in their experiments is 0.768. Whereas, The Dice score with hybrid loss is 0.777. Therefore, it is evident that the use of focal loss gave a boost in Dice score.

Also, Çiçek *et al.* suggested using weighted cross-entropy loss in the V-Net architecture[62]. The Cross validation score on the test set obtained in their 3D network is 0.704. This loss should be tested in future in our experiments.

The F_2 loss has yielded an increase in Dice performance with a value between 0.02 - 0.05 for multi-modality input in all our experiments. The F_2 loss was better in performance than F_1 loss, as shown in the Table 5.1 and 5.7 describing the performance summary statistics on the validation and test set respectively. Thus, introducing generalized F_2 loss in this thesis proved to be beneficial in enhancing the overall performance.

To understand the difference between the Dice loss and the F_β loss for higher values of beta, we have to refer the equation 2.1.12 of the F_β metric. The two losses F_β and Dice are similar except that F_β weighs sensitivity β times more than the precision. Therefore, an increase in performance for higher values of beta signifies that the generalization gap between sensitivity is more significant than that for precision [66] (i.e, performance decrease between the training set and validation/testing set for sensitivity is larger than that for precision.)

6.0.3 Further Recommendations

The process of delineating 3D images involves a high level of expertise while designing the network for segmentation. One has to decide on numerous features, including model design, hyper-parameters, imaging modality, image sizes, and many more [67]. A little modification in any of them can cause a drastic decline in performance. Hence experimenting on all hyperparameter combinations is essential for the model to provide better performance in the delineation process.

The experiments performed in this project have resulted in some recommendations that we were unable to complete due to time limitation and memory that may help boost overall performance in segmenting tumors and lymph nodes in PET/CT images.

Preprocessing

Data augmentation like rotating, mirroring, and elastic deformations should also be tested [68]. Zhu *et al.* [64] and Çiçek *et al.* [62] have suggested that preprocessing, like Data augmentation such as rotation, has given a considerable boost in performance.

Also, the use of Windowing preprocessing is recommended for future projects. It gave an adequate performance in our dataset as described in the Table 5.5 with a window center of 70 HU and a window width of 200 HU. However, other windowing parameters [69] should be tested.

Architecture and Hyper-parameters

The Architectures mentioned above in the section 6.0.1 should all be tested in the future. Dense 3D convolutional neural networks [63], have proven to perform exceptionally well as compare to 3D U-net in many tumor delineation tasks. Also, the loss functions mentioned in the section 6.0.2 should be introduced along with the dense architectures.

Choosing the right optimizer is also very important in the segmentation process. The only optimizer used in the project was Adam, with a learning rate of 0.0001. The other optimizer, like SGD+momemtum, should be tested.

The batch size chosen in our experiments was 2. It is chosen dependent on the available memory, and therefore training network with batch size larger than three on a single GPU gave memory issues.

However, It is preferable to gradually increase the batch size while training compared to reducing the learning rate [70]. This way of increasing the batch size will help in the larger exploration of parameter space in early iterations and precise optimization in later layers.

Dropout regularization with dropout probability 0.5 also has proven to give good results in 3D convolutional networks as suggested by Erden *et al.* [61]. However, we implemented a Dropout regularization with a dropout probability of 0.3 and 0.5 as described in Table 5.3 and Table 5.4. But, unfortunately introducing the dropout regularization does not help in boosting the performance in our model.

6.0.4 Performance comparison with 2D U-Net

Our project's main aim is to compare the performance of the V-Net model with the 2D U-net model [48], on the same head and neck dataset. Yngve Mardal Moe has designed the 2D U-net model.

In the 2D U-net, 3D image data is transformed into slices before it is being fed to the network. Whereas, in the V-Net, the entire 3D volumetric PET/CT images are given to the model for performing experiments.

The average dice score on the test set obtained for the combination of PET/CT images by the 2D U-Net is 0.66. However, on the same test dataset, the average dice score of the V-Net is 0. 6750 which is slightly higher than the 2D U-net performance. The loss functions, filter size, activation functions, and batch normalization, were kept similar in the 2D U-net network.

Hence, it is evident from our experiments that the 3D U-net (V-net) can be a good alternative for the segmentation of the tumor. Also, 3D images contain more spatial feature information about cancer tumors and provide more information than 2D image slices. Therefore, with all the architecture and hyperparameter recommendations, the 3D convolutional neural network's performance can be increased, and these models can become beneficial for the medical image analysis field in the future.

Chapter 7

Conclusion

In this thesis, we implemented a V-Net model using *deoxys* framework for tumor segmentation of 3D PET/CT images of head and neck cancer patients. This project makes use of 3D convolutions operations to take complete advantage of volumetric information for multi-modality images. We successfully created an HDF5 Data-Reader for handling massive image data. The architecture is successfully applied to HNC patients for the automatic segmentation of GTV using the Orion cluster with access to GPU to reduce the memory consumption and computational burden. The project also uses a Tensorboard performance logger.

The highest performing PET/CT model gave F_β score of 0.6750 and Dice score of 0.6286 on the test set. While performing the delineation process, it has been shown that deep learning can be very consistent, time-saving in the medical image analysis field, and for the segmentation of tumors and malignant lymph nodes tissue in HNC patients. The 3D V-net model has shown an adequate performance and can be a preferable choice over the 2D convolution networks. However, our proposed model does not reach the expected dice performance as expected, so we cannot conclude that the automatically generated segmentation maps are similar to those produced by radiologists. Still, deep learning has a vast potential, which can considerably change the way of delineation being observed by radiologist presently and can serve as a second approach in the delineation process.

Bibliography

[1] World Heath Organization, *All cancers fact sheet*, http://www.who.int/news-room/fact-sheets/detail/cancer, Downloaded 2019-02-14, 2018.

[2] C. Wilson, S. Tobin and R. Young, 'The exploding worldwide cancer burden', *International Journal of Gynecologic Cancer*, vol. 14, no. 1, pp. 1–11, 2004.

[3] Baskar, R. Lee K. A.and Yeo, Yeoh and K. W., *Cancer and radiation therapy: Current advances and future directions*, https://doi.org/10.7150/ijms.3635, 2012.

[4] C. K. Kaushal, *Deep learning for automatic tumor delineation of anal cancer based on mri, pet and ct image*, 2019.

[5] FDA Artificial Intelligence, *Regulating the future of healthcare*, https://missinglink.ai/guides/deep-learning-healthcare/fda-artificial-intelligence-regulating-future-healthcare, Downloaded 2020-02-13.

[6] C. Njeh, 'Tumor delineation: The weakest link in the search for accuracy in radiotherapy', *Journal of medical physics/Association of Medical Physicists of India*, vol. 33, no. 4, p. 136, 2008.

[7] F. Chollet, *Deep Learning with Python*. Manning Publication Co., 2018.

[8] S. Raschka and V. Mirjalili, *Python machine learning*. Packt Publishing Ltd, 2017.

[9] National Cancer Institute, *Head and neck cancers*, https://www.cancer.gov/types/head-and-neck, Downloaded 04-04-2020, 2018.

[10] C. E. Cardenas, R. E. McCarroll, L. E. Court, B. A. Elgohari, H. Elhalawani, C. D. Fuller, M. J. Kamal, M. A. Meheissen, A. S. Mohamed, A. Rao *et al.*, 'Deep learning algorithm for auto-delineation of high-risk oropharyngeal clinical target volumes with built-in dice similarity coefficient parameter optimization function', *International Journal of Radiation Oncology* Biology* Physics*, vol. 101, no. 2, pp. 468–478, 2018.

[11] Z. Guo, N. Guo, K. Gong, Q. Li *et al.*, ‘Gross tumor volume segmentation for head and neck cancer radiotherapy using deep dense multi-modality network’, *Physics in Medicine & Biology*, vol. 64, no. 20, p. 205 015, 2019.

[12] T. S. Hong, W. A. Tomé and P. M. Harari, ‘Heterogeneity in head and neck imrt target design and clinical practice’, *Radiotherapy and Oncology*, vol. 103, no. 1, pp. 92–98, 2012.

[13] Q. Song, J. Bai, D. Han, S. Bhatia, W. Sun, W. Rockey, J. E. Bayouth, J. M. Buatti and X. Wu, ‘Optimal co-segmentation of tumor in pet-ct images with context information’, *IEEE transactions on medical imaging*, vol. 32, no. 9, pp. 1685–1697, 2013.

[14] J. Yang, B. M. Beadle, A. S. Garden, D. L. Schwartz and M. Aristophanous, ‘A multimodality segmentation framework for automatic target delineation in head and neck radiotherapy’, *Medical physics*, vol. 42, no. 9, pp. 5310–5320, 2015.

[15] Z. Zeng, J. Wang, B. Tiddeman and R. Zwiggelaar, ‘Unsupervised tumour segmentation in pet using local and global intensity-fitting active surface and alpha matting’, *Computers in biology and medicine*, vol. 43, no. 10, pp. 1530–1544, 2013.

[16] H. Yu, C. Caldwell, K. Mah, I. Poon, J. Balogh, R. MacKenzie, N. Khaouam and R. Tirona, ‘Automated radiation targeting in head-and-neck cancer using region-based texture analysis of pet and ct images’, *International Journal of Radiation Oncology* Biology* Physics*, vol. 75, no. 2, pp. 618–625, 2009.

[17] C. E. Cardenas, B. M. Anderson, M. Aristophanous, J. Yang, D. J. Rhee, R. E. McCarroll, A. S. Mohamed, M. Kamal, B. A. Elgohari, H. M. Elhalawani *et al.*, ‘Auto-delineation of oropharyngeal clinical target volumes using 3d convolutional neural networks’, *Physics in Medicine & Biology*, vol. 63, no. 21, p. 215 026, 2018.

[18] C. Zhang, X. Sun, K. Dang, K. Li, X.-w. Guo, J. Chang, Z.-q. Yu, F.-y. Huang, Y.-s. Wu, Z. Liang *et al.*, ‘Toward an expert level of lung cancer detection and classification using a deep convolutional neural network’, *The Oncologist*, vol. 24, no. 9, pp. 1159–1165, 2019.

[19] A. Chon, N. Balachandar and P. Lu, ‘Deep convolutional neural networks for lung cancer detection’, *Standford University*, 2017.

[20] I. R. I. Haque and J. Neubert, ‘Deep learning approaches to biomedical image segmentation’, *Informatics in Medicine Unlocked*, p. 100 297, 2020, ISSN: 2352-9148. DOI: https://doi.org/10.1016/j.imu.2020.100297. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S235291481930214X.

[21] K. Men, X. C. Y. Zhang, T. Zhang, J. Dai*, J. Yi and Y. L, 'Deep deconvolutional neural network for target segmentation of nasopharyngeal cancer in planning computed tomography images.', *National Cancer Center/Cancer Hospital, Chinese Academy of Medical Sciences and Peking Union Medical College, Beijing, China*, 2017.

[22] J. M. Nazzal, I. M. El-Emary and S. A. Najim, 'Multilayer perceptron neural network (mlps) for analyzing the properties of jordan oil shale 1', 2008.

[23] F. Milletari, N. Navab and S.-A. Ahmadi, 'V-net: Fully convolutional neural networks for volumetric medical image segmentation', pp. 565–571, 2016.

[24] I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[25] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006, ISBN: 0387310738.

[26] K. Pokrass, 'Neural networks – activation functions', 18-10-2019, Downloaded 16-03-2020.

[27] X. Glorot, A. Bordes and Y. Bengio, 'Deep sparse rectifier neural networks', in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011, pp. 315–323. [Online]. Available: http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf.

[28] H. Li, Y. Tian, K. Mueller and X. Chen, 'Beyond saliency: Understanding convolutional neural networks from saliency prediction on layer-wise relevance propagation', *Image and Vision Computing*, vol. 83-84, pp. 70–86, 2019, ISSN: 0262-8856. DOI: https://doi.org/10.1016/j.imavis.2019.02.005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0262885619300149.

[29] M. D. Zeiler and R. Fergus, 'Visualizing and understanding convolutional networks', in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele and T. Tuytelaars, Eds., Cham: Springer International Publishing, 2014, pp. 818–833, ISBN: 978-3-319-10590-1.

[30] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, 'Gradient-based learning applied to document recognition', *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[31] 'Cs231n: Convolutional neural networks for visual recognition', Downloaded 2020-02-20. [Online]. Available: http://cs231n.github.io/convolutional-networks/.

[32] J. Koushik, 'Understanding convolutional neural networks', *arXiv preprint arXiv:1605.09081*, 2016.

[33] S. J. Cruchon-Dupeyrat, 'Deep learning - convolutional neural networks for image classification', 2017, Downloaded 08/03/2020.

[34] A. Deshpande, 'A beginner's guide to understanding convolutional neural networks part 2', July 29, 2016, Downloaded 05/03/2020.

[35] J. T. Springenberg, A. Dosovitskiy, T. Brox and M. Riedmiller, 'Striving for simplicity: The all convolutional net', *arXiv preprint arXiv:1412.6806*, 2014.

[36] V. Dumoulin and F. Visin, 'A guide to convolution arithmetic for deep learning', *arXiv preprint arXiv:1603.07285*, 2016.

[37] H. Gao, H. Yuan, Z. Wang and S. Ji, 'Pixel transposed convolutional networks', *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PP, pp. 1–1, Jan. 2019. DOI: 10.1109/TPAMI.2019.2893965.

[38] Naoki Shibuya, *Up-sampling with transposed convolution*, https://medium.com/activating-robotic-minds/up-sampling-with-transposed-convolution-9ae4f2df52d0, Downloaded 07-04-2020, 2017.

[39] S. Ioffe and C. Szegedy, 'Batch normalization: Accelerating deep network training by reducing internal covariate shift', *arXiv preprint arXiv:1502.03167*, 2015.

[40] S. Santurkar, D. Tsipras, A. Ilyas and A. Madry, 'How does batch normalization help optimization?', pp. 2483–2493, 2018.

[41] I. Gitman and B. Ginsburg, 'Comparison of batch normalization and weight normalization algorithms for the large-scale image classification', *arXiv preprint arXiv:1709.08145*, 2017.

[42] Q. Xu, M. Zhang, Z. Gu and G. Pan, 'Overfitting remedy by sparsifying regularization on fully-connected layers of cnns', *Neurocomputing*, vol. 328, pp. 69–74, 2019.

[43] H. A. Al-Barazanchi, H. Qassim and A. Verma, 'Novel cnn architecture with residual learning and deep supervision for large-scale scene image categorization', in *2016 IEEE 7th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, IEEE, 2016, pp. 1–7.

[44] K. He, X. Zhang, S. Ren and J. Sun, 'Deep residual learning for image recognition', in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[45] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, 'Dropout: A simple way to prevent neural networks from overfitting', *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html.

[46] M. Zinkevich, M. Weimer, L. Li and A. J. Smola, 'Parallelized stochastic gradient descent', in *Advances in neural information processing systems*, 2010, pp. 2595–2603.

[47] D. P. Kingma and J. Ba, 'Adam: A method for stochastic optimization', *arXiv preprint arXiv:1412.6980*, 2014.

[48] Y. M. Moe, 'Deep learning for automatic delineation of tumours from pet/ct images', 2019.

[49] S. Ruder, 'An overview of gradient descent optimization algorithms', *arXiv preprint arXiv:1609.04747*, 2016.

[50] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro and B. Recht, 'The marginal value of adaptive gradient methods in machine learning', in *Advances in Neural Information Processing Systems*, 2017, pp. 4148–4158.

[51] X. Liu, Z. Deng and Y. Yang, 'Recent progress in semantic image segmentation', *Artificial Intelligence Review*, vol. 52, no. 2, pp. 1089–1106, 2019.

[52] Z. Akkus, A. Galimzianova, A. Hoogi, D. L. Rubin and B. J. Erickson, 'Deep learning for brain mri segmentation: State of the art and future directions', *Journal of digital imaging*, vol. 30, no. 4, pp. 449–459, 2017.

[53] V. Badrinarayanan, A. Kendall and R. Cipolla, 'Segnet: A deep convolutional encoder-decoder architecture for image segmentation', *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 12, pp. 2481–2495, 2017.

[54] O. Ronneberger, P. Fischer and T. Brox, *U-net: Convolutional networks for biomedical image segmentation*, Springer, 2015.

[55] A. A. Taha and A. Hanbury, 'Metrics for evaluating 3d medical image segmentation: Analysis, selection, and tool', *BMC medical imaging*, vol. 15, no. 1, p. 29, 2015.

[56] L. R. Dice, 'Measures of the amount of ecologic association between species', *Ecology*, vol. 26, no. 3, pp. 297–302, 1945.

[57] N. Chinchor, 'The statistical significance of the muc-4 results', in *Proceedings of the 4th conference on Message understanding*, Association for Computational Linguistics, 1992, pp. 30–50.

[58] Dr Christopher Lovell, *H5py: Reading and writing hdf5 files in python*, https://www.christopherlovell.co.uk/blog/2016/04/27/h5py-intro.html, Downloaded 09-02-2020.

[59] A. Collette, *Python and HDF5: Unlocking Scientific Data.* " O'Reilly Media, Inc.", 2013.

[60] J. M. Moan, C. D. Amdal, E. Malinen, J. G. Svestad, T. V. Bogsrud and E. Dale, 'The prognostic role of 18f-fluorodeoxyglucose pet in head and neck cancer depends on hpv status', *Radiotherapy and Oncology*, vol. 140, pp. 54–61, 2019.

[61] B. Erden, N. Gamboa and S. Wood, '3d convolutional neural network for brain tumor segmentation', *Computer Science, Stanford University, USA, Technical report*, 2017.

[62] Ö. Çiçek, A. Abdulkadir, S. S. Lienkamp, T. Brox and O. Ronneberger, '3d u-net: Learning dense volumetric segmentation from sparse annotation', in *International conference on medical image computing and computer-assisted intervention*, Springer, 2016, pp. 424–432.

[63] Z. Guo, N. Guo, K. Gong, Q. Li *et al.*, 'Gross tumor volume segmentation for head and neck cancer radiotherapy using deep dense multi-modality network', *Physics in Medicine & Biology*, vol. 64, no. 20, p. 205 015, 2019.

[64] W. Zhu, Y. Huang, L. Zeng, X. Chen, Y. Liu, Z. Qian, N. Du, W. Fan and X. Xie, 'Anatomynet: Deep learning for fast and fully automated whole-volume segmentation of head and neck anatomy', *Medical physics*, vol. 46, no. 2, pp. 576–589, 2019.

[65] J. Hu, L. Shen and G. Sun, 'Squeeze-and-excitation networks', in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7132–7141.

[66] S. S. M. Salehi, D. Erdogmus and A. Gholipour, 'Tversky loss function for image segmentation using 3d fully convolutional deep networks', in *International Workshop on Machine Learning in Medical Imaging*, Springer, 2017, pp. 379–387.

[67] F. Isensee, P. F. Jäger, S. A. Kohl, J. Petersen and K. H. Maier-Hein, 'Automated design of deep learning methods for biomedical image segmentation', *arXiv preprint arXiv:1904.08128*, 2020.

[68] F. Isensee, J. Petersen, A. Klein, D. Zimmerer, P. F. Jaeger, S. Kohl, J. Wasserthal, G. Koehler, T. Norajitra, S. Wirkert *et al.*, 'Nnu-net: Self-adapting framework for u-net-based medical image segmentation', *arXiv preprint arXiv:1809.10486*, 2018.

[69] H. Lee, M. Kim and S. Do, 'Practical window setting optimization for medical image deep learning', *arXiv preprint arXiv:1812.00572*, 2018.

[70] S. L. Smith, P.-J. Kindermans, C. Ying and Q. V. Le, 'Don't decay the learning rate, increase the batch size', *arXiv preprint arXiv:1711.00489*, 2017.

Appendix A

Experiment Structure

Dataset parameters

The `Experiment` class has a definite structure for its input. This input is structured in the form of JSON(JavaScript Object Notation) configuration files. Below, we provide examples of such files that we use, to provide input in a structured manner to experiment class. The first one is the dataset parameters.

```
"dataset_params" : {
    "class_name": "HDF5Reader",
        "config": {
            "filename": "../script/head_neck_new4.h5",
            "batch_size": 2,
            "x_name": "input",
            "y_name": "target",
            "batch_cache": 5,
             "train_folds": [
                4,5,6,12,18,25
            ],
            "val_folds": [
                29,30,31
            ],
            "test_folds": [
                33,
                35,
                37
            ]},
            "preprocessor": {
                "class_name":"WindowingPreprocessor",
                "config": {
                    "window_width": 100,
                    "window_center": 70
                    }
                }
            }
```

This `dataset_parameters` enables the experiment class to extract the dataset from the described HDF5 file using the filename provided. Also describing the batch size , and which fold number to use for training ,validation ad testing containing 3D images.

Train and Input parameters

Once we read the dataset using the HDF5 reader we also have to provide the parameters required during training of dataset such as epochs and a user can also use any automatic performance logger such as Tensorboard and provide the necessary arguments for using it in callbacks of `train_params`. The actual size needed for any input image can be provided using `input_params`

```
 "train_params" :  {
        "callbacks":{
            "class_name": "TensorBoard",
                "config": {
                    "log_dir": "logs",
                    "update_freq":"batch",
                    "profile_batch":  2,
                    "histogram_freq": 1
                }
        },
            "epochs": 100
        }
```

```
"input_params": {
                "shape": [
                173,
                191,
                265,
                2
            ]
        }
```

Here, `train_params` provide the information about how many epochs a user want to set to run the experiments and Tensorboard logger setup in callbacks. A user can provide more than one automatic performance logger in callbacks. `input_params` signifies the standard shape that the dataset posses for any data provided for the experiment.

Model parameters

The `model_params` are used to provide information such as optimizer, loss, performance metric and model architecture for V-Net, which a user wants to use to perform experiments on the dataset. This information is provided to the `Experiment` class utilizing the structure as described below :

```
"model_params": {
            "loss": {
                "class_name": "BinaryFbetaLoss"
            },
            "optimizer": {
                "class_name": "adam",
                "config": {
                    "learning_rate": 0.0001
                }
            },
            "metrics": [
                {
                    "class_name": "BinaryFbeta"
                },
                {
                    "class_name": "Fbeta"
                }
            ]
        },
          "architecture": {
            "type": "Vnet",
            "layers": [
            {
                "name": "conv1",
                "class_name": "Conv3D",
                "config": {
                    "filters": 32,
                    "kernel_size": 3,
                    "activation": "relu",
                    "kernel_initializer": "he_normal",
                    "padding": "same"
                },
                "normalizer": {
                    "class_name": "BatchNormalization"
                }
            },
            {
                "class_name": "MaxPooling3D"
            },

            {
                "name": "conv2",
                "class_name": "Conv3D",
                "config": {
                    "filters": 32,
                    "kernel_size": 3,
                    "activation": "relu",
                    "kernel_initializer": "he_normal",
                    "padding": "same"
                },
                "normalizer": {
                    "class_name": "BatchNormalization"
                }
            },
            {
                "class_name": "MaxPooling3D"
            },
            {
                "name": "conv3",
                "class_name": "Conv3D",
                "config": {
                    "filters": 64,
                    "kernel_size": 3,
                    "activation": "relu",
                    "kernel_initializer": "he_normal",
                    "padding": "same"
                },
                "normalizer": {
                    "class_name": "BatchNormalization"
                }
            },
            {
                "class_name": "MaxPooling3D"
            },
            {
                "class_name": "Conv3D",
                "config": {
                    "filters": 64,
                    "kernel_size": 3,
                    "activation": "relu",
                    "kernel_initializer": "he_normal",
                    "padding": "same"
                },
                "normalizer": {
                    "class_name": "BatchNormalization"
                }
            },
            {
                "name": "conv_T_1",
                "class_name": "Conv3DTranspose",
                "config": {
                    "filters": 64,
                    "kernel_size": 3,
                    "strides": 1,
                    "kernel_initializer": "he_normal",
                    "padding": "same"
                }
            },
            {
                "name": "conv_T_2",
                "class_name": "Conv3DTranspose",
                "config": {
                    "filters": 32,
                    "kernel_size": 3,
                    "strides": 1,
                    "kernel_initializer": "he_normal",
                    "padding": "same"
                },
                "inputs": [
                    "conv3",
                    "conv_T_1"
                ]
            },
            {
                "name": "conv_T_3",
                "class_name": "Conv3DTranspose",
                "config": {
                    "filters": 32,
                    "kernel_size": 3,
                    "strides": 1,
                    "kernel_initializer": "he_normal",
                    "padding": "same"
                },
                "inputs": [
                    "conv2",
                    "conv_T_2"
                ]
            },

            {
                "class_name": "Conv3D",
                "config": {
                    "filters": 1,
                    "kernel_size": 3,
                    "activation": "sigmoid",
                    "kernel_initializer": "he_normal",
                    "padding": "same"
                },
                "inputs": [
                    "conv1",
                    "conv_T_3"
                ]
            }
        ]
    }
```

The architecture defined for V-Net above is an example provided for experiments. A user can increase the number of layers ,number of filters, loss functions and optimizers based on requirement.

Appendix B

Converting dataset into HDF5 format

The first step in the project is to convert the head and neck dataset into HDF5 format. Its very important that all the data should be of same size. The code below shows reading indices of training ,validation and test patient's using different H5 files.

```
# importing all the necessary libraries for creating HDF5 files
import numpy as np
import math
import h5py
from tqdm import tqdm


# used for visual(text based) progress during long running
    operations
def progress_bar(iterable):
    return tqdm(list(iterable))


# Extracting and printing the training(train_indices) , validation(
    val_indices) and
testing(test_indices) from head and neck dataset containing 3D
    images

with h5py.File('../Dataset_3D_U_Net/Pnr.h5', 'r') as Patient_num:
    patient_id_indices = Patient_num['dataset1'][:].squeeze().astype
        (int)

with h5py.File('../Dataset_3D_U_Net/PnrVal.h5', 'r') as val_num:
```

```python
    val_indices = val_num['dataset3'][:].squeeze().astype(int)
    val_new_indices = list(val_indices)
    print(val_new_indices)

with h5py.File('../Dataset_3D_U_Net/PnrTest.h5', 'r') as test_num:
    test_indices = test_num['dataset2'][:].squeeze().astype(int)
    test_new_indices = list(test_indices)
train_indices = list(set(patient_id_indices) - set(val_indices) -
    set(test_indices))
assert len(train_indices) == len(patient_id_indices) - len(
    val_indices) - len(test_indices)
```

The code below will create a function to combine CT and PET images into CT/-PET multi-modality image. The function returns a 4D image tensor with two channels. For example input shape of CT and PET image is (173, 191, 265), after combination using `extract_input_image` function, the output image shape will become (173, 191, 265, 2).

```python
# Method for combining CT and PET images and returns a 4D input
    image tensor with two channels

def extract_input_image(h5):
    ct_image = h5['imdata/CT']
    pt_image = h5['imdata/PT']
    final_image = np.stack([ct_image[:], pt_image[:]], axis=-1).
        squeeze()
    expected_shape = (173, 191, 265, 2)

    if final_image.shape != expected_shape:  # Check for expected
        shape after stacking CT and
        # PET images and reshaping it to desired shape if not
            correct
        print(f"{h5} has shape {final_image.shape}")
        try:
            print("Trying to reshape")
            final_image.reshape(expected_shape)
        except:
            print("Failed to reshape ")
            return None

    return final_image.astype('float32')
```

The final target images are combined using `extract_mask` function which is formed by combination of tumor and lymphatic nodes. The final target is 4D image tensor.

```python
def extract_mask(h5):  # method extract_mask to combine tumor and
    nodes to create the final target
    # (mask) 4D image tensor
    tumor = h5['imdata/tumour'][:].squeeze()
    nodes = h5['imdata/nodes'][:].squeeze()
    return np.logical_or(tumor, nodes).squeeze().astype('float32')
        [..., np.newaxis]
```

The next step is to create two separate lists for storing images and their target masks .using the method `extract_fold_data`.

After this, we create an HDF5 file structure using `create_fold` function where data is divided into folds, which is an HDF5 group containing datasets (images, masks, patient numbers).

```
# method that extract fold data into two separate lists for images
    and masks i.e. 4D images and masks for all the head and neck
    patients.

def extract_fold_data(fold_indices,
                      file_pattern='../Dataset_3D_U_NET/imdata/
                          imdata_P{patient_id:03d}.mat',
                      verbose=True):
    if not verbose:
        def iterate(x):
            return x
    else:
        iterate = progress_bar

    images, masks = [], []
    for patient_id in iterate(fold_indices):
        with h5py.File(file_pattern.format(patient_id=int(patient_id
            )), 'r+') as h5:
            input_image = extract_input_image(h5)
            if input_image is not None:
                images.append(input_image)
                target = extract_mask(h5)
                masks.append(target)

    return images, masks


# method that create folds which are hdf5 group containing dataset
    input,targets and patient numbers of all the 197 head and neck
    patients.

def create_fold(fold_indices, fold_num, filename, verbose=True):
    fold_indices = list(fold_indices)

    if verbose:
```

```python
        print(f"Creating fold {fold_num}")
    images, masks = extract_fold_data(fold_indices)

    # The groups(the different folds) are created with dataset(input
        , targets, and patient number)
    with h5py.File(filename, 'a') as hdf:
        group = hdf.create_group(f'fold_{fold_num}')  # create a
            group with name eg. 'fold_0'
        group.create_dataset('input', data=images, compression='lzf'
            , chunks=True)
        group.create_dataset('target', data=masks, compression='lzf'
            , chunks=True)
        group.create_dataset('pat_num', data=fold_indices)

    return group
```

The last step is to define an HDF5 file where we want to store the training, testing and validation folds. It is named as `head_neck_new4.h5`. This name can be anything of user's choice.

The next step is to divide training, validation, and test patient indices into different folds using the `create_chunks` method. We then define the total number of patients in each fold, which we choose five . Define the total number of patients in train folds, validation folds, and test folds. Finally, iterate over train folds, and store the group into the defined HDF5 file. The same procedure repeated for validation and test set. Just make sure that always check the counting of indices for validation set start from where the training indices end and counting indices for testing set begin from where the validation indices end.

```python
# methods to chunk groups into train ,validation and testing
def create_chunks(length, n):
    # looping till length
    for i in range(0, len(length), n):
        yield length[i:i + n]


patient_each_fold = 5  # choosing the number of patients in each
    fold
train_num_patients = len(train_indices) # total number of patient's
    in training set
val_num_patients = len(val_new_indices) # total number of patient's
    in validation set
test_num_patients = len(test_new_indices) # total number of patient'
    sin test set


fold_num_train = int(np.ceil(train_num_patients / patient_each_fold)
    )  # total no of folds
# for training patient's
fold_num_val = int(np.ceil(val_num_patients / patient_each_fold))  #
     total no of folds
# for validation patient's
fold_num_test = int(np.ceil(test_num_patients / patient_each_fold))
     # total no of folds
# for testing patient's


file_name = "head_neck_new4.h5"  # the name of HDF5 file that will
    be generated with train,
# validation and test folds
num = 0


train_windows = list(create_chunks(train_indices, patient_each_fold)
    ) #splitting training patients
# list with 5 patient each

for i in range(fold_num_train):
    train_fold = create_fold(fold_indices=train_windows[i],
                             fold_num=num, filename=file_name)
    print(train_fold)
    num += 1

# Create folds for val
val_windows = list(create_chunks(val_new_indices, patient_each_fold)
    )
for s in range(fold_num_val):
    val_fold = create_fold(fold_indices=val_windows[s],
                           fold_num=num, filename=file_name)

    num += 1

# Create folds for test
test_windows = list(create_chunks(test_new_indices,
    patient_each_fold))
for t in range(fold_num_test):
    test_fold = create_fold(fold_indices=test_windows[t],
                            fold_num=num, filename=file_name)
    num += 1
```

The final `Experiment` class uses this HDF5 file and the JSON configuration file for running experiments on the head and neck dataset.

Appendix C

Orion Cluster Experiment

It is essential to define a slurm script, and a singularity container while running an experiment on the Orion cluster, as discussed, is section 4.1.6. The below code describes the sample Slurm script used in the project. It contains all the information regarding the memory usage ,number of GPU's to be used and combine the experiment with the methods define in the Singularity container.

```
#!/bin/bash
#SBATCH --ntasks=1               # 1 core(CPU)
#SBATCH --nodes=1                # Use 1 node
#SBATCH --job-name=vnet_test   # sensible name for the job
#SBATCH --mem=196G             # Default memory per CPU is 3GB.
#SBATCH --partition=gpu   # Use the verysmallmem-partition for jobs
    requiring < 10 GB RAM.
#SBATCH --gres=gpu:1
#SBATCH --mail-user=afmi@nmbu.no # Email me when job is done.
#SBATCH --mail-type=ALL
#SBATCH --output=outputs/vnet-%A.out
#SBATCH --error=outputs/vnet-%A.out

# If you would like to use more please adjust this.

## Below you can put your scripts
# If you want to load module
module load singularity

## Code
# Hack to ensure that the GPUs work
nvidia-modprobe -u -c=0

# Run experiment
```

```
echo "Copying data..."
bash copy_dataset.sh
echo "Copy finished"
singularity exec --nv deoxys.sif python experiment.py
```

The Singularity container contains all the necessary libraries, software, scientific workflows needed while running our experiment.

```
Bootstrap: docker
From: tensorflow/tensorflow:latest-gpu-py3
Stage: build

%post
    apt update -y
    apt upgrade -y
    pip install ipython
    pip install http://github.com/huynhngoc/deoxys/archive/master.
       zip
    pip install tensorflow==2.0.0
    pip install comet-ml
    pip install scikit-image
```

More information on running experiment ont the Orion cluster can be found on my Github page: https://github.com/afreenmirza3010/msc-cluster-code.

Thank you.