DD2424 - Deep Learning in Data Science

Kishore Kumar

Disen Ling
disen@kth.se

Zhenghong Xiao
xzhe@kth.se

May, 24 2022

1 Introduction

Heart disease is one of the major diseases that endanger human health, and the study of the heart is an important topic in the medical field. The observation and diagnosis of the heart through medical imaging is an effective method, and with the development of computer and medical imaging technology, magnetic resonance images (MRI) are increasingly used for the diagnosis of heart diseases.

In this project, a neural network is used to segment the left ventricle which can save a lot of time and increase the number of patients a doctor can diagnose compared with the traditional manual segmentation. The U-NET segmentation architecture is used for both binary-segmentation and multi-class-segmentation of heart MRI images. This project tries to replicate the results of the paper "U-Net: Convolutional Networks for Biomedical Image Segmentation"[1]. While the original paper performs segmentation on Electron Microscope images of cells in the Drosophila first instar larva ventral nerve cor, we are using the technique to segment cardiac images instead. To achieve the A grade, we also plan to make the network perform multi-segmentation, to separately detect the left ventricle's two components, the right ventricle, and the background.

2 Model

U-Net is a convolutional neural network that was developed for biomedical image segmentation. The network is based on a fully convolutional network and its architecture was modified and extended to work with fewer training images and to yield more precise segmentation. In this section, the U-net neural network structure will be introduced in detail with its background knowledge and principles.

2.1 Convolutional Network

In deep learning, a convolutional neural network (CNN, or ConvNet) is a class of artificial neural networks (ANN), most commonly applied to analyze visual imagery[2]. A convolutional neural network consists of an input layer, hidden layers, and an output layer. In any feed-forward neural network, the middle layers are called hidden layers because their inputs and outputs are masked by the activation function and final convolution. In a convolutional neural network, the hidden layers include layers that perform convolutions. Typically this includes a layer that performs a dot product of the convolution kernel with the layer's input matrix. This product is usually the Frobenius inner product, and its activation function is commonly ReLU. As the convolution kernel slides along the input matrix for the layer, the convolution operation generates a feature map, which in turn contributes to the input of the next layer. This is followed by other layers such as pooling layers, fully connected layers, and normalization layers. Figure 1 gives an illustration of how the kernel in a CNN layer produces a feature map.

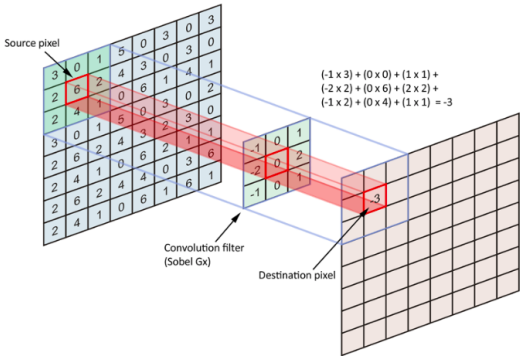


Figure 1: Convolution using a kernel.

2.2 Pooling

Pooling is a computational method of reducing parameters, and the method of reducing parameters is to remove some directly, and the pooling layer is the processing layer dedicated to reducing parameters. Pooling is generally divided into max-pooling and mean-pooling, and the more commonly used one is max-pooling. The specific operation is shown in Figure 2.

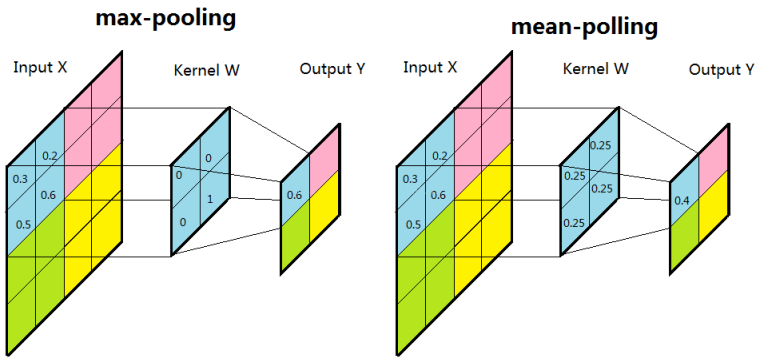


Figure 2: Pooling operation explanation.

The size of the pooling kernel is also designed according to the number of parameters to be removed. Figure 2 shows that the original 4×4 image on the left is pooled with a 2×2 pooling kernel and then reduced to a 2×2 image.

2.3 Up sampling

The so-called upsampling is to recover the image from a lower size [C,H,W] to a larger size [C,sH,sW], where s is the upsampling multiplier. This process is also known as full convolution. Whether it is semantic segmentation, object detection, or 3D reconstruction models, when the extracted high-level features need to be scaled up, it is necessary to upsample the feature map. The common methods for upsampling are the nearest interpolation, bilinear interpolation, bi-cubic interpolation, transposed convolution, and un-pooling. The upsampling can be simply represented by Figure 3.

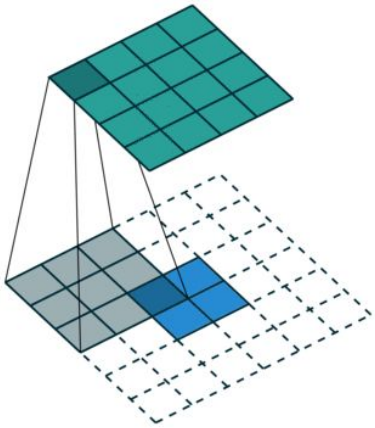


Figure 3: Up sampling.

2.4 Skip connections

U-Net was mainly used for medical image segmentation, which required refined segmentation results, such as image segmentation of fine structures like blood vessels. In the past, the classical framework for image segmentation was the fully convolutional network and coder-decoder framework. In a fully convolutional network, one module acts as the encoder and one module acts as the decoder. The information of the image will be greatly compressed in the middle by the encoder and finally, a series of deconvolution or upsampling operations by the decoder are used to get the final segmentation result.
Obviously, the deconvolution or upsampling process needs to fill in a lot of gaps and generate something from nothing, and this process lacks enough auxiliary information. The advantage of using skip-connection is that the feature information at the corresponding scale from the encoder block is introduced into the upsampling or deconvolution process, which provides multi-scale and multi-level information for the later image segmentation, and thus a finer segmentation result can be obtained. Figure 4 illustrates the skip connections present in the U-NET architecture.

2.5 Model architecture

The architecture of the U-NET model is shown by Figure 4.

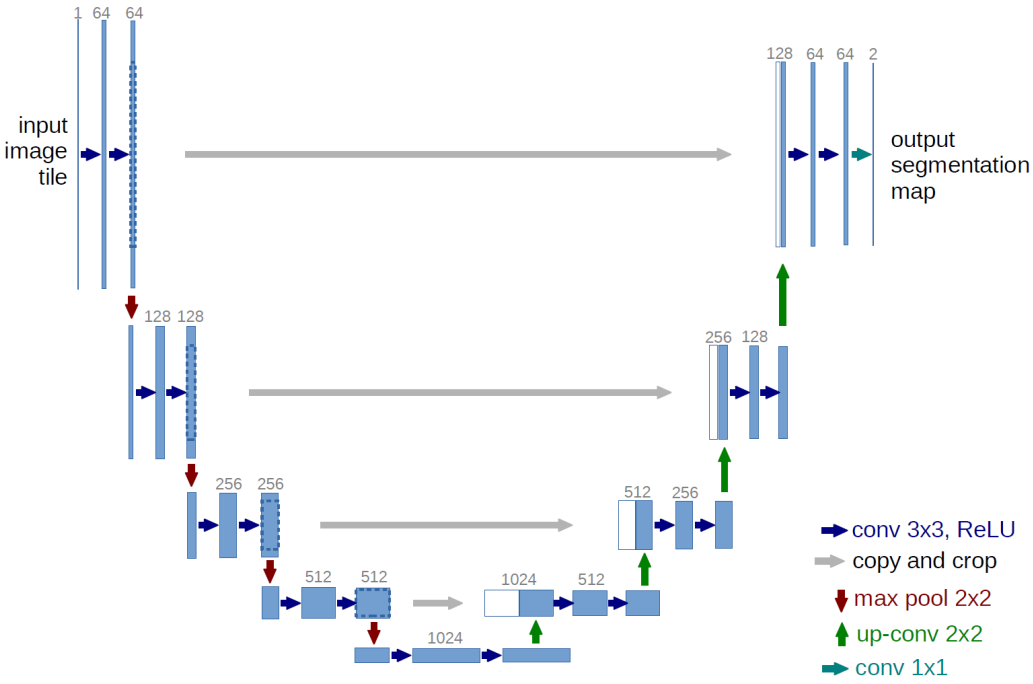


Figure 4: U-NET architecture.

This U-net network has four layers, which down sample and up sample the images 4 times each.

In the encoder on the left of the U-net, the input image will be down-sampled all the way down into a 16x16x1024 feature matrix starting from an input dimension of 256x256x1 (binary segmentation) through a series of convolution and max-pooling operations. And the 16x16x1024 feature matrix will be up-sampled into 32x32x512 at the beginning of the right part by a 512 channel 2x2 up-sampling kernel. And then the feature matrix is concatenated with the previous 32x32x512 through a skip connection to 32x32x1024. Then 32x32x1024 is converted to 32x32x512 by a 512 channel 3x3 kernel followed by batch norm and ReLU. In the remaining steps, the feature matrix will be up-sampled into 256x256x 64 from 32x32x512 just like a reversed version of the encoder. Finally, this 64 channel 256x256 input is convolved with a Convolutional layer with only 1 channel to produce a 1 channel output pf the dimension 256x256x1. As we know, the input also has 1 channel. So basically we get an output that shares the same dimensions as the input.

The above process applies specifically for binary segmentation. Slight variations exist for Multi-Class Segmentation. For binary segmentation, the input is a one-channel greyscale image and the output is a one-channel output that is passed to the sigmoid activation function and the binary cross entropy loss is computed by PyTorch's Binary cross-entropy with logits loss (BCEWith-

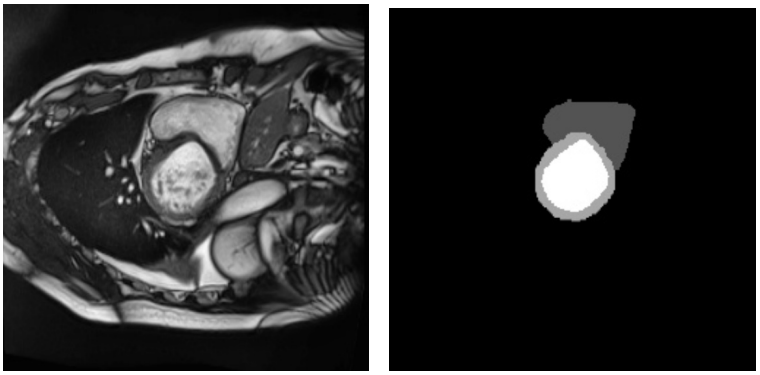
LogitsLoss) function. Here both the neural network's output, as well as the predicted output, have only one channel. For multi-segmentation, the input is a three-channel RGB image and the output is a four-channel output. The four-channel output means that each output pixel has four values that correspond to four segments of the heart: background, right ventricle, left ventricle boundary, and left ventricle itself. These four values are passed to a Softmax function and then the four segments' class probabilities for each pixel are obtained.

3 Dataset

In this section, the dataset used in the project will be described in detail. And we will describe the source of the dataset and detail what we have done to process it.

3.1 Introduction

Cardiac MRI Image Dataset from ACDC Challenge of MICCAI 2017 will be used in this project to train the neural network. The dataset consists of a total of 1808 images out of which 1710 images will be used for training 98 images will be used for testing, and 5 percent of the training images will be used for validation. Figure 5 shows an example of the input MRI image and the segmentation mask of the cardiac MRI image. Theses images will be resized and augmented based on the requirements of the neural network. In the segmentation mask, the different segments of the heart are indicated by different shades of grey. Multi-class segmentation will also be included in this project and the data set is the same as the one used to train the binary segmentation network. The difference is that the target images will be pre processed into a binary image for binary segmentation and retained in its original form for multi-class segmentation.



(a) Original MRI Images (b) Cardiac Segmentation

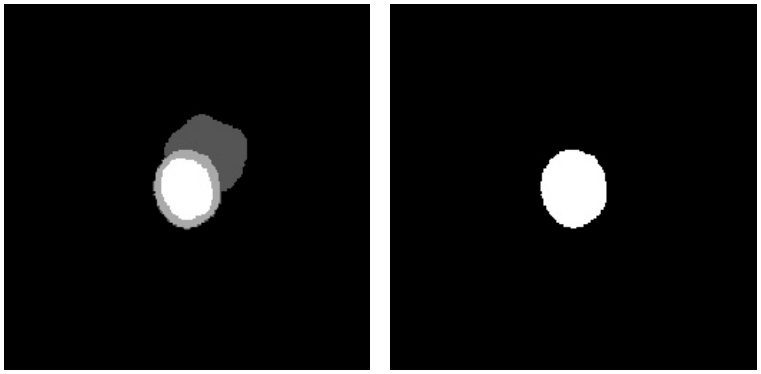
Figure 5: The Cardiac MRI Image Dataset

3.2 Labels

In this section, the labels or the ground truth images used to train the network will be introduced. And the binary-segmentation and multi-segmentation labels will be introduced separately.

3.2.1 Binary-segmentation

For the binary segmentation, the cardiac MRI images should be segmented into 2 parts: the background and the left ventricle. So the original cardiac segmentation images should be processed to get binary cardiac segmentation images. The white and light gray parts of the original image will be merged together and represented in white to represent the left ventricle. And the black and dark gray parts of the original image will be merged together and represented in black to represent the background. The processed images will be used as the label for the binary segmentation. And the label of binary segmentation is a 1 channel label which means each pixel has only 1 value: 1 or 0. Figure 6 has shown this process below:



(a) Original segmentation Image (b) Binary Segmentation Image

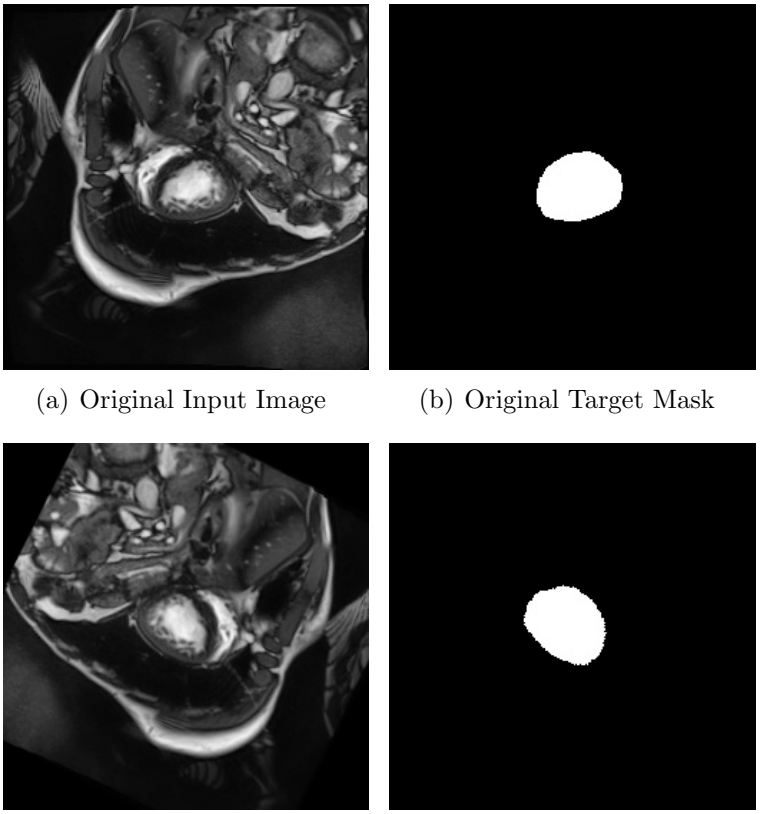
Figure 6: Turn the multi-segmentation labels into binary-segmentation labels

3.2.2 Multi-segmentation

For the Multi-segmentation, the cardiac MRI images should be segmented into 4 parts: background, right ventricle, left ventricle boundary, and left ventricle itself. So the original cardiac segmentation images shown in Figure 5 can be used without processing. It is worth mentioning that the label for multi-segmentation is a 1 channel label instead of a 4 channel label. This means each pixel has only 1 value. These values are either 0,1,2 or 3 for the 4 corresponding segments. This 1 channel target class is one-hot encoded by PyTorch during the training process.

3.3 Augmentations

Due to the limitation of the training set, we need to perform Augmentations of the training part of the data. In the process of inputting the training set, we randomly rotate some MRI images in the dataset and their corresponding label images by a certain angle to obtain new data to augment the dataset. Similarly, the input training images will be flipped horizontally and Vertically and with a certain probability to achieve the purpose of augmenting the dataset. We have chosen to rotate the images and their respective masks with a random angle between -30 and +30 degree, horizontally flip the image and the mask with a probability of 0.5 and vertically flip them with a probability of 0.1. Figure 7 illustrates an example where an input image and its corresponding segmentation map have been rotated by a random angle and flipped horizontally.



(a) Original Input Image (b) Original Target Mask

(c) Augmented Input Image (d) Augmented Target mask

Figure 7: Augmentation of the input and target images.

3.4 Libraries

To process the images, we need to import some image processing libraries. So, the OpenCV library and the Albumentations library are imported to load, pre-process, resize and augment the images of dataset.

4 Training the Neural Network

4.1 An overview

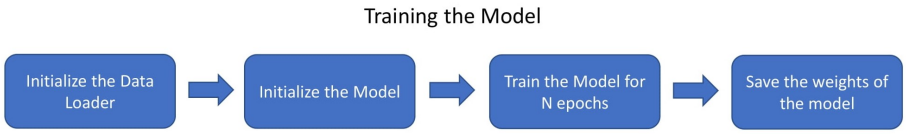


Figure 8: A general overview of training a neural network.

The training of the neural network is depicted in figure 8. Following the definition of the data loader class with the augmentations mentioned in section 3.3, a data loader object is defined, which is responsible for augmenting the training input images and providing them as a batch, along with the appropriate labels. The images and their labels are flipped, rotated, and resized during the augmentation process.

The neural network is then developed when this stage has been completed. The PyTorch framework, created by Facebook's (now Meta) Artificial Intelligence division, was used to do this. The PyTorch framework's nn sub module includes various capabilities that make describing neural network architecture easier. As a result, the U-NET neural network design described in section 2.5 is implemented in Python as class. After that, an object for the model architecture class is initialised, which contains the fully convolutional neural network architecture's weights, biases, activation functions, skip connections, and so on. This is the neural network that we must train and then utilize to get inferences from the testing data.

After that, the model is trained for as many epochs as necessary, and the trained weights and biases of the neural network are saved as a .pth file. This model may be loaded into any PyTorch program, avoiding the need to train the model with data each time it is used.

4.2 Feed forward and back propagation

Figure 9 depicts the stages involved in training a neural network with training data for one epoch. Based on the batch size of the data, the training dataset is divided into N batches. The neural network is trained with these N batches for 1 epoch, then retrained with these batches for the following epoch. Because data augmentation is stochastic in nature, the same batches

will not exist for the next epoch. The images are also randomly sampled for each batch.

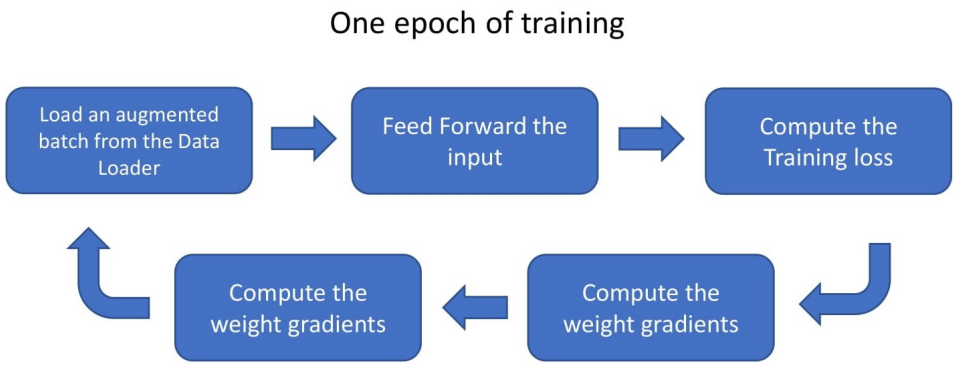


Figure 9: Training a neural network for one epoch.

For every training step, the augmented input and labels of a batch are loaded by the data loader object. The input of the neural network is fed through the model and the neural networks predictions are recorded. Ideally, the predicted values should be equal to the target labels, however, that is not the case and thus the difference between the targets and the predictions is calculated using a loss function.

For our project we have used two different loss functions for Binary Segmentation and Multi class segmentation accordingly. The **BCEWithLogitsLoss** or the Binary cross entropy with logits loss function of the Pytorch module is used for Binary segmentation. The formula for this loss has been provided in figure 10. Where x_n is the neural network's n^{th} predicted output and y_n is the n^{th} target value of the batch. Ideally x_n should be equal to y_n. The target is a 1 channel image with each pixel being either 0(background) or 1(left ventricle). Each 1 channel output predicted by the neural network is passed into a sigmoid activation function and the binary cross entropy loss if computed by comparing it with the 1 channel target pixel class (either 0 or 1) which is specified by the target label.

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n)) \right],$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \text{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

Figure 10: Binary Cross Entropy with logits Loss of PyTorch.

For multi-class segmentation the **NLLLoss** or the negative log likelihood loss of the PyTorch module is used. The formula for this loss is provided in figure 11. Here an additional parameter w_c is used. w_c is a weight value which can be assigned to each class. Since we are using a balanced dataset, w_c is set as 1 and can be ignored. The target is a 1 channel image with each channel being either 0(background), 1 (right ventricle), 2(left ventricle boundary) or 3(left ventricle). The soft max activation function in the output layer should ideally predict 1 in the neuron corresponding to the pixel's label and predict 0 in the other neurons. This 4 channel output where each class probability of a pixel is being predicted by the softmax activation function is compared with the target pixel class specified by the 1 channel target label. Although the predicted output is a 4 channel output, the target is a 1 channel label. But PyTorch automatically one hot encodes these target pixel classes and computes the Negative Log Likelihood Loss.

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} x_{n,y_n}, \quad w_c = \text{weight}[c] \cdot 1\{c \neq \text{ignore\_index}\},$$

where x is the input, y is the target, w is the weight, and N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{y_n}} l_n, & \text{if reduction} = \text{'mean';} \\ \sum_{n=1}^{N} l_n, & \text{if reduction} = \text{'sum'.} \end{cases}$$

Figure 11: negative Log Likelihood Loss of PyTorch.

Once the loss has been computed, the gradients of the weights of the neural network with respect to the loss is computed. The auto-grad module of PyTorch is used to achieve this. Once the gradients are computed, the weights of the model are updated using them. We have chosen the Adam Optimiser to perform the weight update step due to its versatile performance. Figure 12 explains the Adam Optimizer algorithm. This algorithm has several parameters. *params* is the iterable of parameters to optimize. γ is the learning rate specified by the user. β_1 and β_2 are the coefficients used for computing the running averages of gradient and its square. ϵ is the term added to the denominator to improve numerical stability. λ is the L2 regularisation or the weight decay component. *amsgrad* parameter specified whether to use the AMSGrad variant of this algorithm from the paper "On the Convergence of Adam and Beyond "[3]. *maximize* maximizes the *params* based on the objective, instead of minimizing.

By repeating the above process for every batch, the model generalises and gets better at segmenting the heart images. At the end of each epoch, the validation loss if computed. If the validation loss obtained at the current epoch is lower than the lowest validation loss obtained so far, it means that

input : γ (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective)
λ (weight decay), *amsgrad*, *maximize*
initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

for $t = 1$ **to** ... **do**
  if *maximize* :
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t / (1-\beta_1^t)$
  $\widehat{v_t} \leftarrow v_t / (1-\beta_2^t)$
  if *amsgrad*
    $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \left(\sqrt{\widehat{v_t}^{max}} + \epsilon\right)$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \left(\sqrt{\widehat{v_t}} + \epsilon\right)$

return θ_t

Figure 12: Adam Optimiser Algorithm

the model has generalised better. Validation is performed in evaluate mode. More details about the evaluate mode is provided in section 5.1

Although visualising the output gives a very good idea of how good the model works, it is a qualitative way of observing the performance of the model. A rather quantitative approach to find the accuracy of the model is to compute an accuracy score. The accuracy scores used for binary segmentation and multi-class segmentation are the *Sørensen–Dice coefficient/Dice Score* and *Pixel classification Accuracy* respectively.

The Dice score is used to measure the similarity between two tensors which for our purposes consists of the Neural network's predicted output and the target labels respectively. The idea is very similar to the concept of intersection over union. The Dice Score is a value between 0 and 1. The score is 0 if there is no overlap between the predicted binary mask and the target mask and the score is 1 if there is a 100% overlap between the predicted mask and

the target mask. An ideal model should produce a dice score of 1 for both the training and testing data. However, real world models couldn't achieve 100% accurate predictions yet. The formula for Sørensen–Dice coefficient is given in figure 13

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

Figure 13: Sørensen–Dice coefficient/Dice Score

where X and Y are the tensors to be compared. The union of the two tensors is obtained by multiplying both the target and the predicted class of the neural network. This way pixels which are correctly classified as 1 generates a product of 1 with the target pixels class and pixels which are wrongly classified as generates a product of 0 with the target pixel class. Thus when all the pixels are wrongly classified, the union product becomes 0.

The pixel classification accuracy is used to measure the accuracy of the multi-class segmentation algorithm. Dice score suits well for binary segmentation but multi-class segmentation requires 4 different dice scores computed for all the 4 different classes. Thus the overall accuracy of the model is instead verified by finding the average ratio between the number of pixels which were correctly classified as either 0,1,2 or 3 and the total number of pixels present in the image for all the images and multiplying this value by 100 to get it in percentage form. Thus the pixel classification accuracy is a number between 0% and 100% with an accuracy of 0% when every pixel is classified wrongly and an accuracy of 100% when the predicted output is exactly the same as the target label mask. Current models are not capable of achieve the maximum accuracy yet. The formula for the Pixel classification accuracy is given in figure 14

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

Figure 14: Pixel Classification Accuracy

During training, the accuracy scores for binary and Multi-Class segmentation are computed at the end of each epoch. If the accuracy score obtained at the current epoch is higher than the highest accuracy score obtained so far, it means that the model has obtained the best accuracy so far and this model is saved. This is known as check pointing. The best check pointed model is then used for obtaining inferences on the testing images in the testing script.

5 Obtaining Inferences from the Testing Data

5.1 A general overview

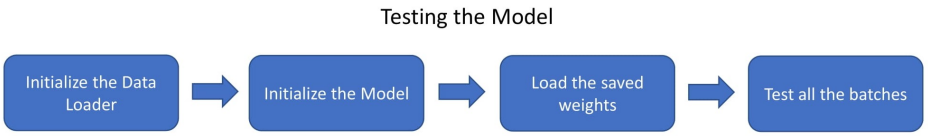


Figure 15: Overview of Testing the network

Once the model has been trained on the training data and a reasonable validation accuracy is achieved, it is time to test the model. figure 15 gives an overall idea of what happens during testing. Similar to training the model, initially, the data loader and the neural network objects are initialised. However, unlike training, the saved weights of the trained model are loaded into the neural network object using PyTorch's load_state_dict() function.

Now that we have the trained model, the testing data is passed through the neural network to obtained the inferences. The testing process varies from the training process, which is explained in detail in the section 5.2. One important thing to be noted is that while obtaining inferences, the neural network is switched to evaluation mode. This ensures that batch normalisation parameters are no longer updated and the trained mean and standard deviation parameters are used for obtaining inferences. This approached is followed for validation as well.

5.2 Feed Forward and Post Processing

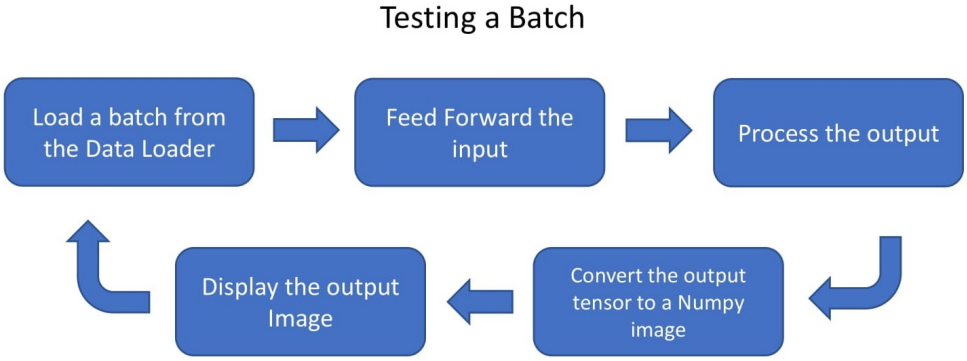


Figure 16: Obtaining and visualising the output of the neural network

Figure 16 details the testing process of the trained neural network. As we know, training the neural network involves the feed forwarding step as well as the back propagation step. However, the testing process involves only the feed forwarding step as the neural network doesn't learning anything new from the test data. The output of the neural network is present in the form of a tensor, which is the datatype used by the PyTorch library. This datatype is incompatible with the OpenCV module. This, we need to convert the tensor back to a numpy array in the *(rows,columns)* and *(rows,columns,channel)* formats for binary segmentation and multi-class segmentation respectively. The numpy() function of the Torch module is used to achieve this.

Now that we have the processed output, the image is displayed using the cv2.imshow() function of the OpenCV module. Examples of the results can be found in section 6.3.

6 Results

6.1 Hyper parameters

Surprisingly, there aren't a lot of hyper parameters to be manually tuned by the user since PyTorch has some default values for most hyper parameters. Thus, only a handful of parameters need to be tuned. The hyper-parameter values are the same for both Binary Segmentation and Multi-class Segmentation. The hyper parameters chosen and the reasoning for them have been documented below:

- *Validation Data split:* **5**% of the training data is used for validation purposes. That is, the neural network does not train on these data. The reason this specific value was chosen was to allow a reasonable large amount to be used for training while having enough data to actually validate the model and make sure the model doesn't over fit.
- *Batch Size:* Each batch contains **4** images and their respective labels. One of the main reason why a small batch size was chosen was due to the fact that the GPU used to train the model was not able to hold vast amount of data at a any moment and thus a relatively smaller batch was chosen. The other reason is that the neural network is observed to generalise faster with smaller batch sizes as compared to larger batch sizes. However, there weren't any pronounced effect of the batch size on the overall accuracy of the training as Adam optimiser is not susceptible to these changes.
- *Learning Rate* The learning rate of the Adam Optimizer was chosen as **0.0001**. This is because the learning rate has to be slow enough to

learn smoothly while fast enough to reach the global minimum loss at a reasonable pace.

- *Epochs:* The model is trained for **30** epochs. This allows the model to train just enough that it reaches its global minimum while not over fitting too much. Even if the model over,fits the concept of check pointing is used to save the generalised models which is explained in detail at the end of section 4.2.

6.2 Training and validation results

6.2.1 Binary Segmentation

For binary segmentation, the neural network was trained with the architecture which receives a single channel input and produces a single channel output. The training losses, validation losses and the Dice Scores have been graphed in figures 17, 18 and 19 respectively.

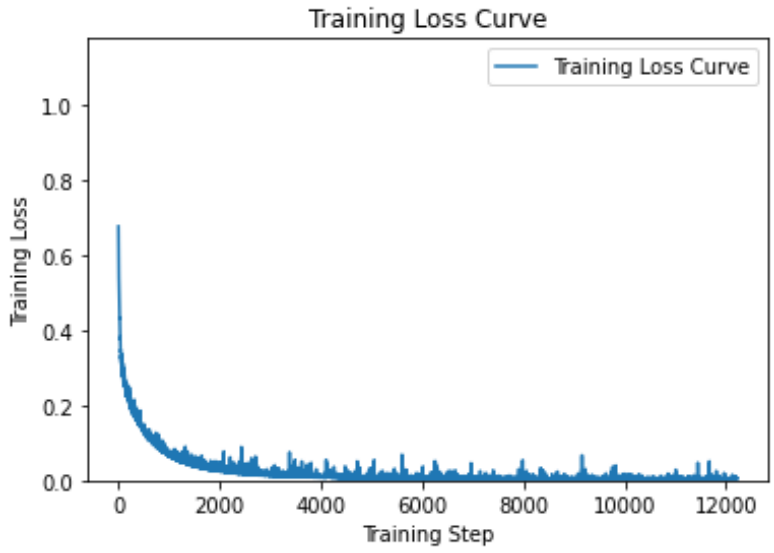


Figure 17: Training Loss of the Binary Segmentation network over 30 epochs (12210 training steps).

It could be noticed that the training loss falls rapidly at the beginning and then falls slowly from epoch 5 on wards (training step 2035). We could also notice that there is a lot of oscillations in the training loss curve. The training loss increases by a very small amount before decreasing due to the fact that the Adam Optimizer has momentum incorporated into it. After 30 epochs, the model achieves a training loss of 0.0033.

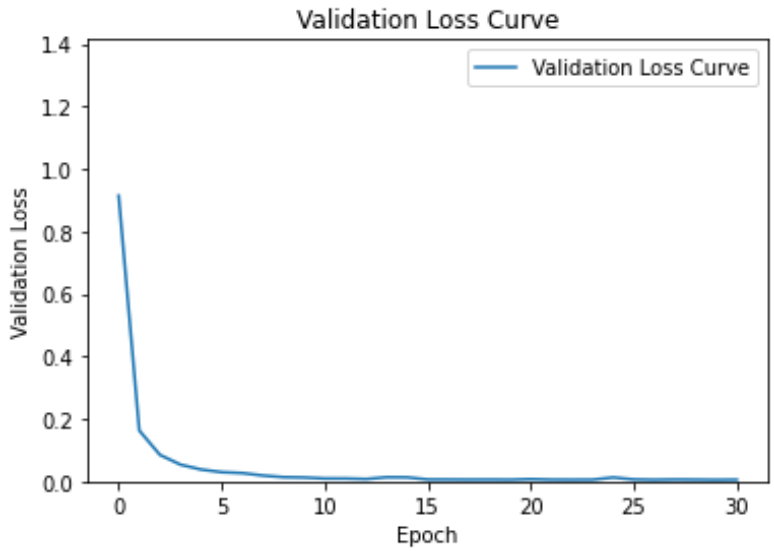


Figure 18: Validation Loss of the Binary Segmentation over 30 epochs.

The validation loss follows a similar trend as the training loss. Notice that the validation loss is computed at the end of each epochs instead of each training step to accelerate the training process. We could notice that the validation loss falls rapidly in the first 5 epochs and then proceeds to fall slowly. It is surprising to note that the validation loss does not increase by a lot even after 30 epochs indicating that there is no over fitting taking place. This indicates that the model has generalised pretty good, which is reflected in the output presented in section 6.3. The model achieves its lowest validation loss of 0.00468 on Epoch 29 which was the last time when the model was check pointed.

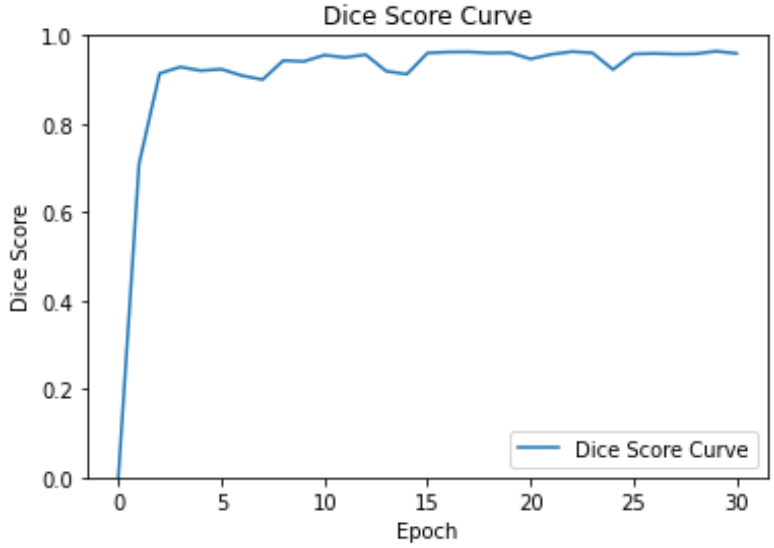


Figure 19: Dice Scores of the Binary Segmentation network over 30 epochs.

We know that accuracy is inversely proportional to the loss of the model. That is, when the loss of the model decreases, the accuracy increases. This is reflected in the Dice coefficient graph. The Dice coefficient takes a huge leap in the first few epochs and the proceeds to increase slowly. The best Dice score - 0.9642 was achieved in epoch 29 which was when the model was finally check pointed. This was the model can be used to test the data. However a more powerful model trained on Google Colab is used for testing the algorithm.

6.2.2 Multi-Class Segmentation

For Multi-Class Segmentation, the neural network was trained with 3 channel RGB input images and a 4 channel output map which contains the class probabilities of each pixel. The model is trained with a 1 channel target image containing the class value of each pixel. This target is one hot encoded into a 4 channel map while training. After training the model for 30 epochs, the results displayed in figure 20, 21 and 22 were obtained.

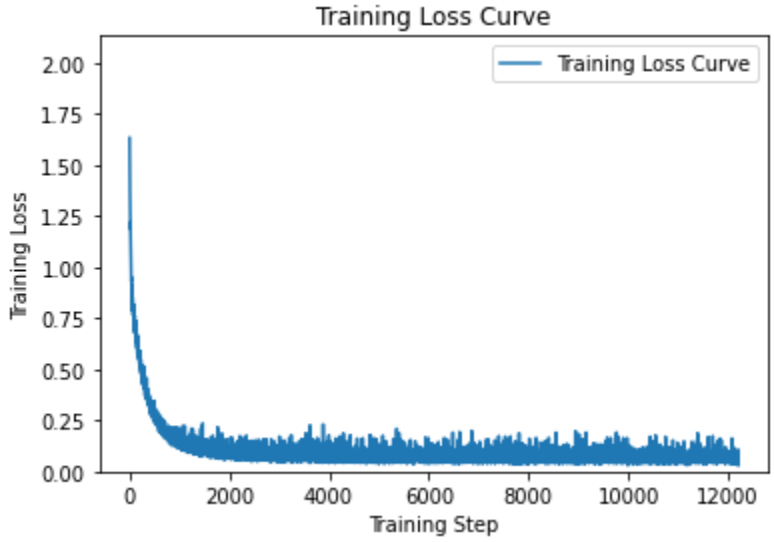


Figure 20: Training Loss of the Multi-Class Segmentation network over 30 epochs (12210 training steps).

The training loss graph is very similar to the Binary Segmentation graph. This is expected as the architecture is more or less the same and the same hyper parameter values are used. After 30 epochs, the model achieved a training loss as low as 0.302.

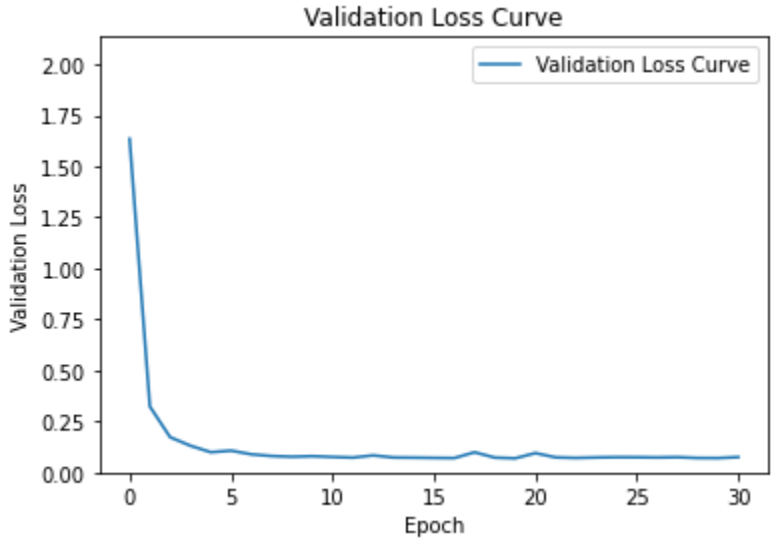


Figure 21: Validation Loss of the Multi-Class Segmentation over 30 epochs.

The validation loss graph shows that over-fitting didn't take place. The model achieved the best validation loss value of 0.0704 on epoch 29. This is the same as the result obtained for binary segmentation.

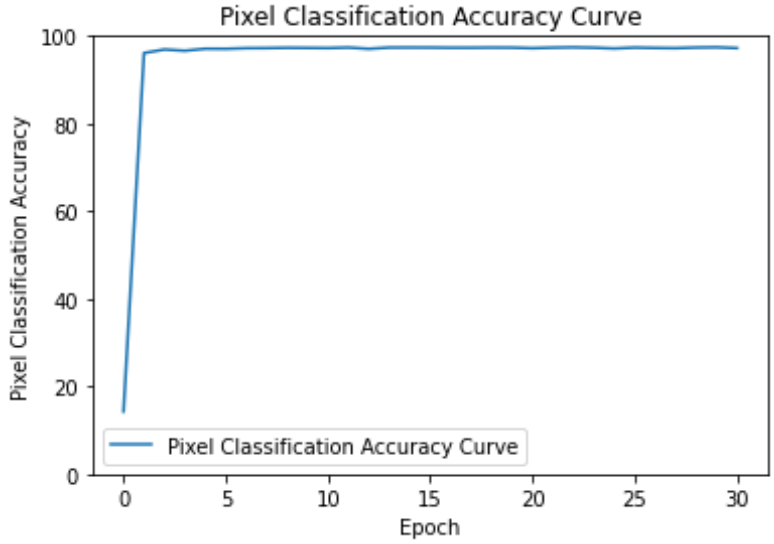


Figure 22: Pixel Classification Accuracy of the Multi-Class Segmentation network over 30 epochs.

The Pixel classification accuracy graph shows that the accuracy is almost 100%. While this might be very encouraging, the pixel classification accuracy is a slightly deceiving metric as a huge fraction of the pixels in the image constitutes the background. Thus a model can achieve a reasonably high accuracy of around 70% by simply classifying all the pixels as background.

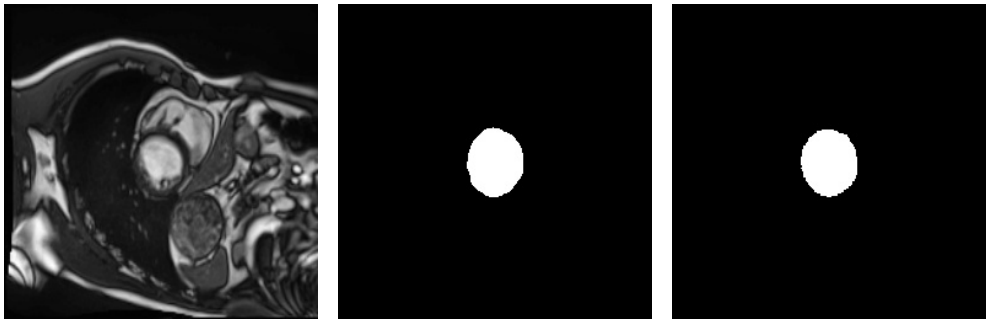
However, the model achieves an accuracy of 97.4358% on the 29^{th} epoch which indicates generalisation. The model was check pointed for the last time on this epoch. This model can be used for testing the neural network. However a more powerful model trained on Google Colab is used for testing the algorithm.

6.3 Testing output

In this section, the performance of the trained network on the test dataset will be shown. Both binary-segmentation network and multi-segmentation network have a good performance on the test dataset. The models trained above were trained on Google Colab. However, the models used for testing are slightly more accurate than the models saved in the training processes described above as shown in the following segments. These models were trained for more epochs on a slightly more powerful GPU on Google Colab. Thus, testing accuracies which are slightly higher than the validation accuracies can be observed.

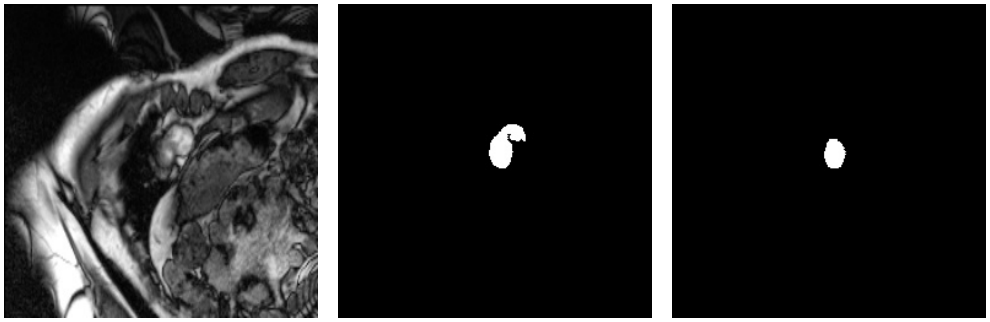
6.3.1 Binary-segmentation

For the binary-segmentation part, the trained network is able to accurately segment the cardiac MRI images into the left ventricle and the background where the white part is the left ventricle and the black part is the background. As shown in the figure 23, the trained network is able to segment the input heart image accurately into the left ventricle and the background. However, in images obtained towards the end of the MRI scan, the model's accuracy suffers as shown in figure 24. But given how hard it is even for a human to distinguish the components in these images, the results are justified. The network got a test Dice score of 0.9474



(a) Input (b) Output (c) Ground truth

Figure 23: A good example of binary-segmentation test result

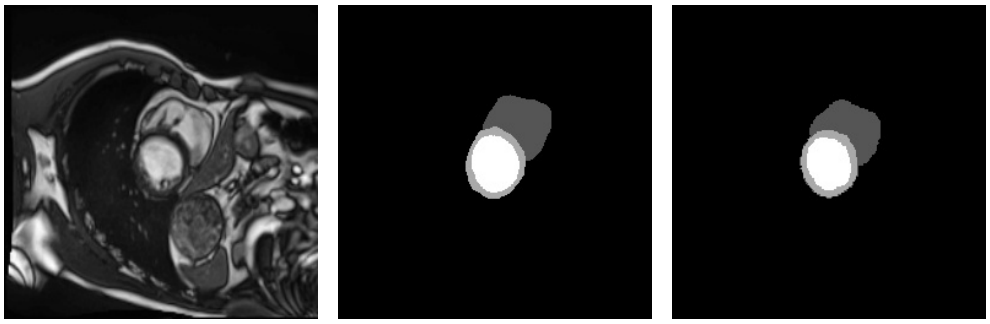


(a) Input (b) Output (c) Ground truth

Figure 24: A bad example of binary-segmentation test result

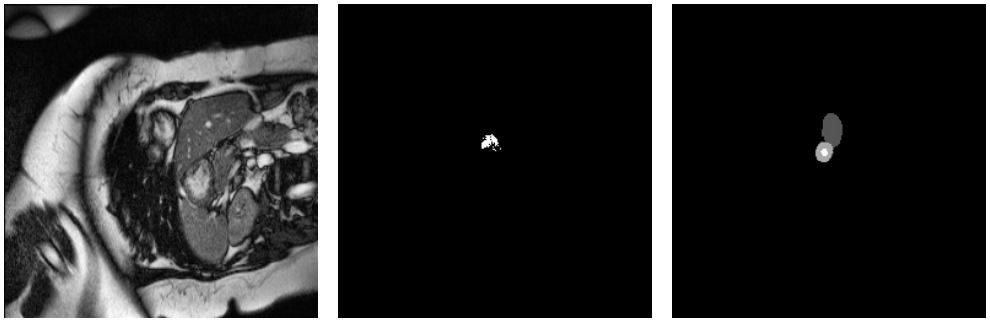
6.3.2 Multi-segmentation

For the multi-class segmentation part, the trained network is able to accurately segment the cardiac MRI images into four parts: the left ventricle's two components, the right ventricle and the background. As shown in the figure 25, the network segments the input MRI image into four parts. And the network can get a test pixel classification accuracy of 98.3342%. Just like Binary Segmentation, the neural network suffers in images obtained during the end of the MRI scan as show in figure 26, but this is expected as the components are not clearly detailed in the input image.



(a) Input (b) Output (c) Ground truth

Figure 25: A good example of Multi-segmentation test result



(a) Input (b) Output (c) Ground truth

Figure 26: A bad example of Multi-segmentation test result

7 Conclusion

Thus, the U-NET architecture was implemented from scratch using the PyTorch framework and the model was trained on MRI Cardiac Images. With successful results obtained from binary segmentation, the Multi-Class U-NET mode, which is a variant of the binary segmentation model was also trained and similar results were obtained. The different losses and accuracy scores have also been graphed and the outputs have been visualised. Near perfect accuracy scores were obtained on both cases with a few errors observed in edge cases. Maybe the concept of LSTM or transformers can be used to extract temporal information between subsequent images to segment the images obtained towards the end of the MRI scan. This is a good area to begin future research

References

[1] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*. Ed. by Nassir Navab et al. Cham: Springer International Publishing, 2015, pp. 234–241. ISBN: 978-3-319-24574-4.

[2] Coenraad Mouton, Johannes C. Myburgh, and Marelie H. Davel. "Stride and Translation Invariance in CNNs". In: *Artificial Intelligence Research*. Ed. by Aurona Gerber. Cham: Springer International Publishing, 2020, pp. 267–281. ISBN: 978-3-030-66151-9.

[3] Sanjiv Kumar Sashank J. Reddi Satyen Kale. "On the Convergence of Adam and Beyond". In: (2019).