Automatic Right Ventricle Segmentation in Cardiac MRI images

Machine Learning Nanodegree Capstone Report

Akshar Kumar

July 2018

1 Definition

1.1 Project Overview

Segmentation of the right ventricle (RV) in patients is a crucial step in diagnosing debilitating diseases. RV segmentation is used when diagnosing diseases such as pulmonary hypertension, coronary heart disease, and cardiomypathies, among others [1]. Currently, the accepted gold standard for evaluating RV volumes is a trained physician performing manual contours on cardiac MRI images. When segmenting the RV, the physician will create an endocardium and epicardium contour of the ventricle. Endocardium refers to the inner wall of the ventricle while epicardium refers to the ventricles outer wall. Figure 1 shows both types of contours. The physician will create contours for consecutive images resulting in a set of 2D contours that when combined, can form a 3D volume of the ventricle.

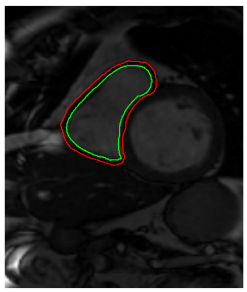


Figure 1: An example of endocardium and epicardium contours of the right ventricle. The green contour is the endocardium contour (inner layer), while the red contour is the epicardium contour (outer layer). This image was provided with the training dataset.

Since there is usually a large number of images to contour, the overall process of segmentation becomes very lengthy. Each image can take anywhere from 10-15 minutes. Aside from the large time cost, there are some difficulties that can make the examination task more subjective:

- fuzziness of borders due to blood flow
- presence of wall irregularities
- complex shape of the RV - it can appear triangular when viewed longitudinally and crescent-shaped when viewed along the short axis
- variability in cine MRI equipment, institutions, and populations
- noise associated with cine MRI images [1, 2]

Because of these difficulties, the process is prone to intra- and inter-observer variability [1]. The purpose of this project was to simplify these complex pain points and provide a method to automatically segment the RV in cardiac MRI images. Such an automated method would decrease segmentation time and improve

consistency of results between physicians. The automated model would then assist the physician in making better diagnoses for the patient.

The dataset used for this project was provided in the Right Ventricle Segmentation Competition conducted in 2012 [1]. For more information about the dataset, see Section 2.1.

1.2 Problem Statement

In this project, we will apply a deep learning model, more specifically a Convolutional Neural Network (CNN), to automatically segment the RV in cardiac MRI images. Segmentation of the RV is useful in characterizing the ejection fraction of the heart. We will compare the predictions of the model to contours created by physicians with years of experience. These manual contours will be what we call the ground truth and serve as the labels for our training data. This specific problem was presented as a computer vision challenge at the International Conference on Medical Image Computing and Computer Assisted Intervention in October 2012.

The solution we will implement is a convolutional neural network based on the U-Net architecture [3]. The U-Net architecture is a popular architecture used for biomedical image segmentation. The specifics of the U-Net architecture are described in Section 2.3. The tasks that we completed during this project can be broken down into three categories:

1. Data Preprocessing
    - Cleaned the data by rotating images to uniform size and loaded only the labeled images.
    - Explored the data by generating the image histogram.
    - Normalized images to reduce heavy bias towards background pixels.
2. Building and training the model
    - Created a model following the U-Net architecture.
    - Split the data set into training and validation data
    - Augmented the training data.
    - Trained on training set with validation for each epoch.
    - Tuned hyper-parameters: dropout, batch normalization, number of epochs, batch size, etc.
3. Testing the model
    - Predicted contours from the testing data.
    - Converted the segmentation map to list of points defining contour boundaries.
    - Submitted these predictions to moderators for evaluation.

1.3 Metrics

For evaluating our model, we used the Dice coefficient. The Dice coefficient is a measure of the overlap between two contours. The coefficient varies from 0 to 1, with a value of 0 indicating no overlap between two contours. Meanwhile, a value of 1 indicates a perfect overlap between two contours. We compared our automated contours to a manual contour performed by an expert physician using the following equation:

$$D(A, M) = 2 * \frac{A \cap M}{A + M} \tag{1}$$

as described by [2]. Where D represents the Dice coefficient, A represents the area of the automated contour, and M represents the area of the manual contour performed by the expert.

[1] `http://www.litislab.fr/?projet=1rvsc`

2 Analysis

2.1 Data Exploration

The dataset used for this project was from the Right Ventricle Segmentation Competition conducted in 2012. The training dataset contains images from 48 patients: 16 training cases, 32 testing cases. For each patient there were a total of 200-280 images. The images provided in each case were 2D cine images with approximately 8-12 continuous, short-axis images spanning a single cardiac cycle for each patient. The cardiac images had been zoomed and cropped to a size of 216x256 pixels. The labels provided were manual RV segmentations of these images, performed by an expert. The processing time per patient was around 15 minutes. There were a total of 243 labeled images in our training dataset. The testing set contained 514 labeled images without manual contours. We submitted our contour predictions on the test images to the moderators for final evaluation. The images are in the Digital Imaging and Communications in Medicine (DICOM) format. DICOM is a communication protocol and file format generally used in medical imaging. It can store medical information along with the patient's information in one file. Since the DICOM format for the data is complex, we used a suggested library, pydicom to work with the images. This library allowed us to load the images into python structures for use in feeding into our model.

Upon exploring the image data, we noticed that data for some patients was rotated 90 degrees, so instead of being 216x256, the images were 256x216. When loading the images into memory to use for training, we rotated all the images to dimensions 216x256. The labels provided were epicardium (outer wall) and endocardium (inner wall) contours. The set of patients images that were contoured were listed in a text file. Using this file, we only loaded images into memory that had corresponding contours. The contours were given as text files with [X,Y] pairs corresponding to the pixel in the image that was part of the RV boundary. When loading the images, we also made sure to rotate contours for images that were not of the 216x256 size. The contours were converted to binary masks using OpenCV.

Another issue with the training set was the number of labeled images available. Because the labeled images available were so few (only 243 for training), we had to perform some data augmentation before training. We introduced random rotations, zooms, and transformations when feeding the images into our model. The augmentation was performed using the `ImageDataGenerator` API in Keras.

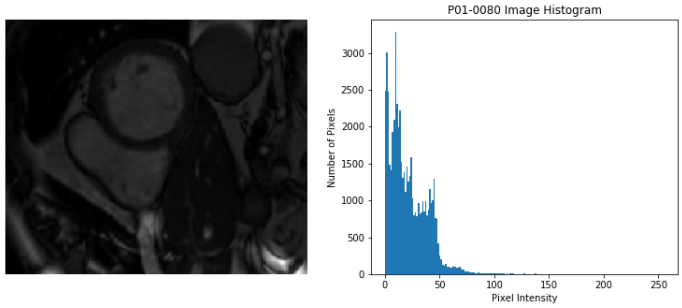
2.2 Exploratory Visualization

Since we were working with an image segmentation problem, we had to first gain some understanding about the images we would be working with. One of the best ways to explore images is to create an image histogram to understand how pixel intensity values are distributed across the image. In order to create an image histogram, we used the Matplotlib library. Figure 2a shows the histogram of one of the training images without any preprocessing performed. From this histogram, we can see that the pixel value intensities are skewed to the right, leaning heavily towards what would be considered the background pixels. This could introduce some issues when we go to train our model with the input images. Instead of having the neural network learn the optimal parameters for the input layer, we might get end up teaching the network optimal weights that are heavily biased towards background pixels.

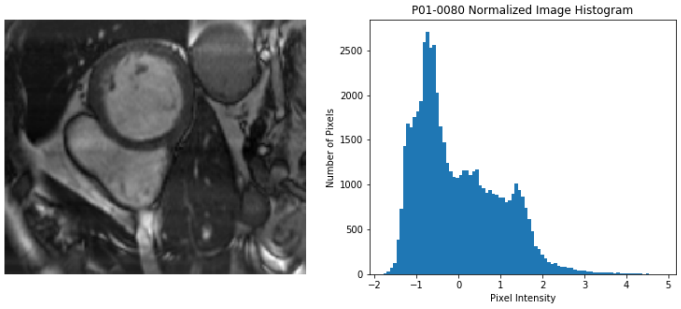
In order to reduce the variance in the data, we normalized the example image by subtracting each pixel value from the image mean and dividing by the image standard deviation. Figure 2b shows the histogram after of that image after being normalized using the technique described. Compared to the raw image histogram, normalizing the image brings the pixel intensity values within a smaller range centered around 0. When we go to train our neural network, having the normalized inputs will allow the network to learn optimal weights faster and across the varying pixel intensity distributions. Exploring the characteristics of our data helped with understanding subtler ways to preprocess the images for improved training. After this exploration, we will normalize all the images in our training set before feeding them into our model.

2.3 Algorithms and Techniques

The technique used to solve this image segmentation problem was a convolutional neural network (CNN). A CNN is a supervised learning model that can learn features and perform robust classification of an image without extensive preprocessing [2]. The CNN model is considered a state-of-the-art algorithm for image classification tasks. The standard architecture for a CNN consists of a convolution layer, a nonlinear activation function, a pooling layer, and a fully connected layer [4]. At each layer, the input is scanned by a filter



(a)



(b)

Figure 2: Examples of (a) a raw image histogram and (b) a normalized image histogram for one of the images in our training set.

which will detect some type of feature. This can be something as simple as an edge in the the early layers, to something more complicated like a face in deeper layers. Since CNNs are very successful in computer vision and image classification problems, they appear to be a natural fit for application in biomedical imaging segmentation [5].

There are many common CNN architectures such as VGGNet, GoogLeNet, and ResNet that are applied to image classification problems [2]. One popular architecture used for biomedical image segmentation is the U-Net architecture, proposed by Ronneberger et al [3]. In this architecture, there is a downsampling path that follows the same structure as a generic CNN. There are convolutional layers followed by max pooling layers, with each step halving the overall image space. The architecture then introduces an upsampling path, which is needed to create a segmentation map of the input image with similar dimensions. In order to create the map, the output from the convolutional layers right before max-pooling during the downsampling path is concatenated with the output of the corresponding up-convolution layer. Using the information from the concatenation, the network is able to re-learn features that were lost from max-pooling [6]. In this way we get a segmentation map that will tell us whether or not a pixel is part of our region of interest. Figure 3 shows this architecture in more detail. We also decided to use this architecture because the authors were able to successfully train a model with only 30 images of labeled data. Since we also have few labeled images, we believe this architecture will provide us with a good start for our model.

When training the CNN, there are certain parameters that we initialized and tuned to improve performance. Some of the training parameters that we adjusted include:

- Number of epochs: length of the training.
- Batch size: how many images to use for a single training epoch.
- Learning rate: how fast the algorithm learns.

In addition to the hyper-parameters listed above, we adjusted the overall architecture of the CNN. This involved changing the number of layers, as well as what type of layers we used in implementation.

In order to use the data for training our CNN, we loaded the images and contours into memory. We used an 80-20 training-validation split for our model. The CNN loaded a batch of images for training at each epoch, and then loaded the validation images to test the accuracy of the model. The batch size was a parameter we configured in the CNN during implementation.

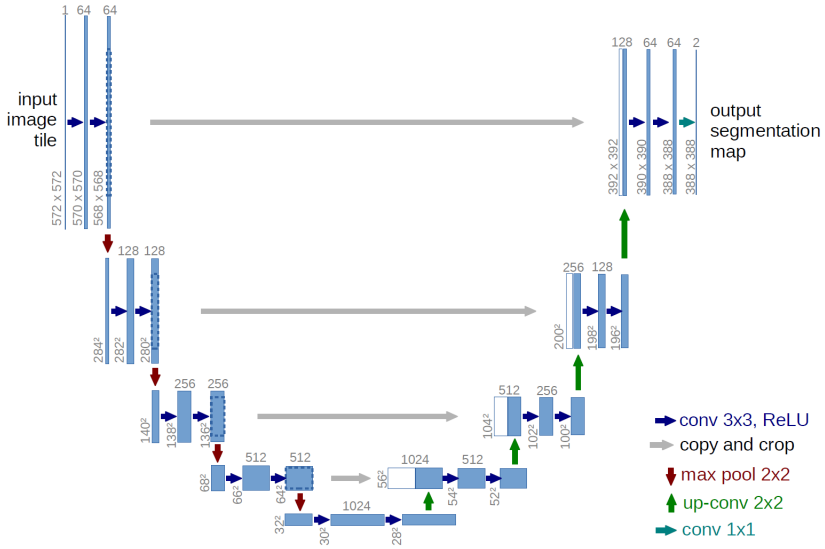


Figure 3: An example of the U-net architecture. Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations. [3]

2.4 Benchmark

The benchmark model we compared our model to was the fully convolutional neural network created by Phi Vu Tran for the RVSC competition [2]. This model is a 15-layer deep fully convolutional neural network (FCN). The FCN comprised of 15 stacked convolutional layers and three max-pooling layers. Each of the convolution layers used a Rectified Linear Unit (ReLU) as the activation function. The total number of parameters was around 11 million. The output of this model was a heat map that predicted the class membership of each pixel in the images. In order to compare the proposed model to this benchmark model, we will use the Dice coefficient, as described in Section 1.3. For our benchmark model, the Dice coefficient for endocardium contours created from the test set was 0.84 with a standard deviation of 0.21. The average Dice coefficient for epicardium contours was 0.86 with a standard deviation of 0.20. We will also compare our model to human level performance. The average Dice coefficient for a human is estimated to be around 0.90(0.10) [1].

3 Methodologies

3.1 Data Preprocessing

The dataset we used in this project was fairly clean. One issue that had to be resolved was rotating all the images to the same height and width. Most of the images in the dataset were 216x256, however there a few images for patients that were 256x216, or rotated 90 degrees. We rotated the images while loading them into memory. In addition to rotating the images, we also had to rotate their corresponding contours. This was done when the image masks were being created from the contour data and loaded into memory.

As described in Section 2.1 we also normalized the images before feeding them into the model. In order to accomplish this normalization, we subtracted the mean pixel value of the entire image and subtracted that from the specific pixel location. Then we divided the difference by the standard deviation of the entire image. We did this for all the training images before performing any further data preprocessing.

To solve the main issue about our small data set, we used data augmentation techniques. We used the `ImageDataGenerator` object in Keras to perform our augmentation. The parameters we selected for augmentation were `rotation_range`=180 degrees, `width_shift_range`= 0.1, `height_shift_range`=0.1, `shear_range`=0.1, `zoom_range`=0.01, and `fill_mode`='nearest'. The image set was split into 80% training and 20% validation sets. The training set was then used for augmentation. Since the masks had to be augmented with their corresponding training images, we created two `ImageDataGenerator` objects, one for the training images and one for the training masks. We ensured that the same augmentation was performed on the image and mask

by passing in the same seed parameter value in the `flow()` function for our generators. We augmented the data before each epoch using the `fit_generator()` function from the Keras API.

3.2 Implementation

The implementation of the model involved the following steps:

1. Data loading: reading images and contours into memory, converting contours to binary masks, and setting up the `ImageDataGenerator` object for data augmentation.
2. U-Net CNN model: coding the downsampling block, upsampling block, and final activation layer with Keras.
3. Prediction class: coding a class to make predictions on the testing sets provided, convert binary masks to contours, and generate submission text files for evaluation.

3.2.1 Data Loading

In order to load the data into memory and serve it as input for training the model, we had to create a custom Python class named `ImageData`. The `ImageData` class loaded the patient images from the DICOM files, created binary masks from the contours provided, and loaded the masks into memory. To extract images from the DICOM files provided, we used the pydicom library. The library has a `read_file()` function that loads the MRI image, and returns it as a Numpy array. The contours were provided as text files with [X,Y] coordinates corresponding to the pixel that was part of the RV boundary. To create these masks from the contours, we used the Pillow library to create a new Image object and then to create a polygon with the same outline as the contour points given. The area of the shape defined by that polygon would contain white pixels while the pixels outside that region would be black. This is how we created the binary masks from the contours given. We stored these masks in a list which was accessed from an instance of the `ImageData` object. In addition to loading the images and masks, we had to rotate some of the images. Most of the images were 216x256, however there were a few patients that had images that were 256x216. In those cases, we had to rotate the images to be 216x256. In addition to rotating the images, we had to rotate the corresponding contours.

3.2.2 CNN implementation

After creating the data loading class, we had to build our convolutional neural network. The neural network architecture we followed was the U-Net structure. The U-Net model was made of a downsampling block followed by an upsampling block to create a segmentation map of similar dimensions as the input image. The downsampling block was a series of 2 convolutional layers followed by a max-pooling layer. The convolutional layers used the ReLU activation function, 3x3 filter size, 32 initial filters, and padding. The max-pooling layer had a pool size of 2. At each downsampling step, the feature size was doubled, and this block of convolution, convolution, max pooling was repeated for 3 steps. At each step, we were also making copies of the output from the second convolution to be used for our upsampling block. When upsampling, we had an up-convolution step followed by two regular convolutions. At each step we concatenated the up-convolution output with the copy of the output from the corresponding downsampling convolution layer. The output of the model was a binary segmentation map of the same dimensions as the input. This was created by applying one last convolution with filter size 2, kernel size 1, and a Softmax activation function. The output map labeled a white pixel as part of the right ventricle, while a black pixel was part of the background. Figure 4 shows the structure of our initial U-Net neural network.

We implemented the U-Net CNN using the deep learning library, Keras, on a Tensorflow backend. The downsampling block used the standard `Conv2D` object for our convolutional layers. We used a `MaxPooling2D` object for our max-pooling layers. For the upsampling block of the network, we had to perform a transposed convolution followed by two `Conv2D` layers. To perform the transposed convolution, we used the `ConvTranspose2D` object from Keras. The code for the `get_unet()` which creates the U-Net model, is shown in Listing 1.

```
1 def get_unet(self, height, width, channels, features=64, steps=4, dropout=0.0,
      padding='same'):
2         layer = Input(shape=(height,width,channels))
3         inputs = layer
```

```
        copies = []
        # downsampling block
        for i in range(steps):
            layer = Conv2D(filters=features,kernel_size=3,padding=padding)(layer)
            #layer = BatchNormalization()(layer)
            layer = Activation('relu')(layer)
            layer = Dropout(dropout)(layer)

            layer = Conv2D(filters=features,kernel_size=3,padding=padding)(layer)
            #layer = BatchNormalization()(layer)
            layer = Activation('relu')(layer)
            layer = Dropout(dropout)(layer)

            copies.append(layer)
            layer = MaxPooling2D(pool_size=(2, 2))(layer)

            features *= 2

        layer = Conv2D(filters=features,kernel_size=3,padding=padding)(layer)
        #layer = BatchNormalization()(layer)
        layer = Activation('relu')(layer)
        layer = Dropout(dropout)(layer)

        layer = Conv2D(filters=features,kernel_size=3,padding=padding)(layer)
        #layer = BatchNormalization()(layer)
        layer = Activation('relu')(layer)
        layer = Dropout(dropout)(layer)


        # upsampling block
        for i in reversed(range(steps)):
            features //= 2
            layer = Conv2DTranspose(filters=features,kernel_size=2,strides=2)(layer)
            crop_copy = self.crop(layer,copies[i])
            layer = Concatenate()( [layer, crop_copy] )

            layer = Conv2D(filters=features,kernel_size=3,padding=padding)(layer)
            #layer = BatchNormalization()(layer)
            layer = Activation('relu')(layer)
            layer = Dropout(dropout)(layer)

            layer = Conv2D(filters=features,kernel_size=3,padding=padding)(layer)
            #layer = BatchNormalization()(layer)
            layer = Activation('relu')(layer)
            layer = Dropout(dropout)(layer)


        layer = Conv2D(filters=2,kernel_size=1)(layer)

        layer = Lambda(lambda x: x/1.0)(layer)
        outputs = Activation('softmax')(layer)

        return Model(inputs=inputs,outputs=outputs)
```

Listing 1: Function from the `UNet` class that creates the U-Net model.

3.2.3 Training

After creating the `ImageData` class and the `UNet` class, we were ready to train our model. All the training was done within a Jupyter Notebook. In order to train the U-Net model, we followed the following steps: Training implementation:

1. Load images and masks into memory.
    - This was done as described using the `ImageData` class.
2. Split images and masks into training and validation sets
    - We split the training set provided into 80% training data and 20% validation.

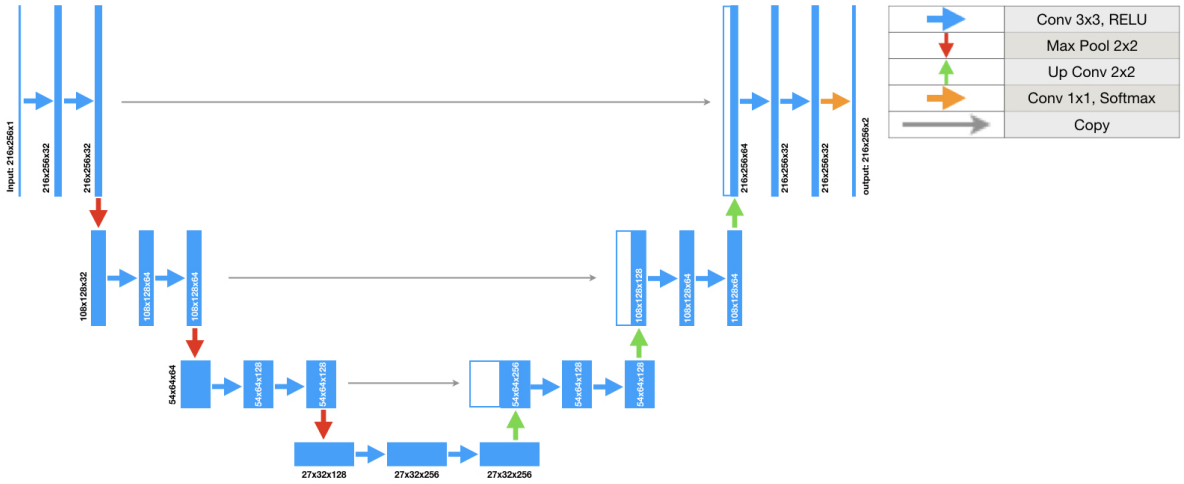


Figure 4: Architecture of the U-Net model that was implemented to solve our segmentation task. Each layer contains the dimensions of its output.

3. Augment training data using `ImageDataGenerator` object with initial parameters as described in Section 3.1.

    - The training data set was augmented by introducing random rotations, shifts, shears and zooms. We created an `ImageDataGenerator` object for both the training images and the training masks. We did this because if an image was augmented, we would have to augment the corresponding mask to serve as the appropriate label for our neural network layers. In order to synchronize the generator objects, we had to provide the same `seed` parameter value in the `flow()` function.

4. Load the model architecture.

    - To load the U-Net convolutional neural network, we created an instance of the `UNet` class and called its `get_unet()` function. The parameters we passed to this function were `height`=216, `width`=256, `channels`=1, `features`=32, `steps`=3, `dropout`=0.0, and `padding`='same'.

5. We trained the model using the `fit_generator()` function and training images and masks generators as input data. We trained for 500 epochs and a batch size of 32. We also passed the validation images and corresponding masks to the `fit_generator()` function as the validation_data parameter. We saved the weights corresponding to the best validation loss value to a weights file. The training was performed on both endocardium images and epicardium images.

6. Calculate the average training and validation Dice by predicting the trained model on training images and validation images using `predict()` function.

    - To evaluate the performance of the model immediately, we used the model to predict on training and validation images and then found the average dice value for each set. The official evaluation was performed by sending the segmentation results on the testing data to the RVSC moderators.

3.2.4 Testing

We were given two test data sets with 32 total patients. In order to evaluate our model on the testing set, we had to load the weights from our trained model and make predictions on each testing image. Then we used the `findContours()` function from the OpenCV library to create a contour from the output binary mask. We set the parameters of the `findContours()` function as `image`=output_mask, `mode`='cv2.RETR_LIST', and `method`= 'cv2.CHAIN_APPROX_NONE'. The contour was then converted to a set of coordinates representing the x-pixel and y-pixel of each point in its boundary. These sets of coordinates where then written to text files and sent to the RVSC moderators for evaluation on the manual labels.

There were some edge cases that had to be handled. The model would sometimes predict two regions of interest for an image. To handle this case we chose the region that contained the most points and selected

that as our predicted contour for evaluation. The other case we had to handle was when no RV was predicted in the image. For this case, we passed an empty array and generated an empty file with no points.

3.3 Refinement

As described above, the U-Net architecture was used to perform automatic segmentation of the right ventricle in MRI images. However, there were multiple iterations and tuning steps performed on the basic U-Net parameters to achieve the best score on our validation set. The results described in Section 4 will be for the model that made predictions on the testing sets, which were then submitted to the contest moderators for final average Dice score.

Throughout the refinement process, we used the Dice score calculated by making predictions on the validation set as an indicator of model accuracy. This allowed us to quickly improve the model without having to constantly submit results on the testing data and wait for a response. All the models described will have the same basic architecture (32 initial features, 3 steps for upsampling and downsampling, and a batch_size of 32). The hyper-parameters that were tuned were the learning rate, number of epochs, dropout, and batch normalization.

The first combination of hyper-parameters that was used for training was 500 epochs, no dropout, and the Adam optimizer with a 0.001 learning rate. After training, we ran predictions on the validation set for immediate feedback on the accuracy of our model. For each image in the validation set, we predicted the mask using our model and then calculated the Dice coefficient. We then averaged the Dice coefficient for all images in the validation set. For the initial model trained, the average training Dice score for endocardium contours was 0.90(0.11) and for the validation set it was 0.76(0.28). When creating epicardium contours, our model achieved an average training Dice score of 0.93(0.08) while the average validation Dice was 0.78(0.29). The difference between the training and validation dice scores for both types of contours suggests that the model is overfitting. We attempted to introduce a dropout value to decrease the overfitting, but this led to poor performance on the validation set. We also added batch normalization layers to the model, but this also reduced the average validation Dice score. For example, the average training Dice score decreased from 0.90 to 0.82 for endocardium images when trained with batch normalization, and the validation Dice score decreased from 0.76 to 0.67. Dropout and batch normalization created worse performance while also not reducing the overfitting. Thus, we had to find another solution to improve the performance of our model.

We noticed the model was incorrectly predicting multiple right ventricles in a single image. Figure 5 shows a prediction where the model predicted multiple right ventricles. One possible reason for this false positive result could be that the segmentation masks were created by upsampling layers that had no view of the entire input image at the deepest levels. For example, after the downsampling layers, the deepest layers in our U-Net had an input size of 27x32. This image space was too small for the neural network to have a global view of the original image.

Figure 5: An example of the regular U-Net incorrectly predicting multiple right ventricles in a single image.

In order to maintain a global context at the deep layers of our network, we introduced dilated convolution layers between the downsampling and upsampling arms of the U-Net model. A dilated convolution layer contains the regular filters which stride across the input space, but the filters are expanded by a dilation factor. Dilated convolutions are beneficial when working with models that are producing dense predictions (i.e. each pixel needs to be classified) [7]. In our case, each pixel of the input image was being classified as either part of the right ventricle or part of the background. Thus, dilated convolutional layers would be a good method to test for performance improvements in our neural network. In implementation, the weights in the filters used for convolution are matched to distant elements of the input, rather than adjacent elements. Figure 6 shows a representation of the filters that include a dilation factor greater than 1.

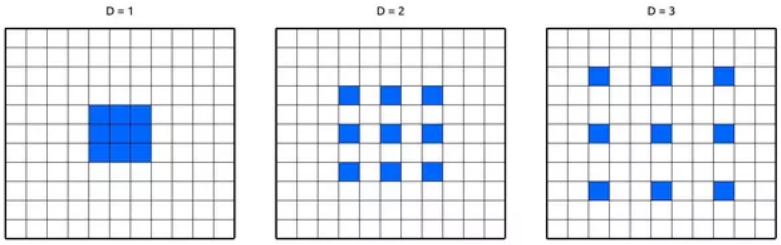


Figure 6: An example of filters that contain dilation factors of 1, 2, and 3. When D=1, we have the standard filter where each element of the filter is 1 position away from the next. When D=2, each element of the filter is 2 elements away. The positions between the dilated elements are set to 0 [8].

Implementing a dilation rate was simple thanks to the Keras API. We increased the number of convolution layers between the downsampling and upsampling blocks from 2 to 4, and gave them dilation rates of 2, 4, 8, and 16. Figure 7 shows the architecture of the U-Net model with dilated convolution layers. We trained the dilated U-Net model for 500 epochs, with no dropout, using an Adam optimizer with a learning rate of 0.001. After training the dilated U-Net model on our data set, we achieved an average training Dice of 0.90(0.11) on endocardium images, and 0.93(0.07) on epicardium images. The average validation Dice for endocardium images was 0.80(0.25) and 0.86(0.17) for epicardium images. We received similar results on the training Dice between both of our models, but the validation Dice was higher in both image types for the dilated U-Net. The dilated U-Net model was still overfitting, but it was less than the regular U-Net model. Figure 8 shows dilated U-Net model's prediction on the same image from Figure 5. We can see that introducing the dilated convolutions improved the ability of our model to have a global context of the input image at its deepest layers, and thus reduced the false positives.

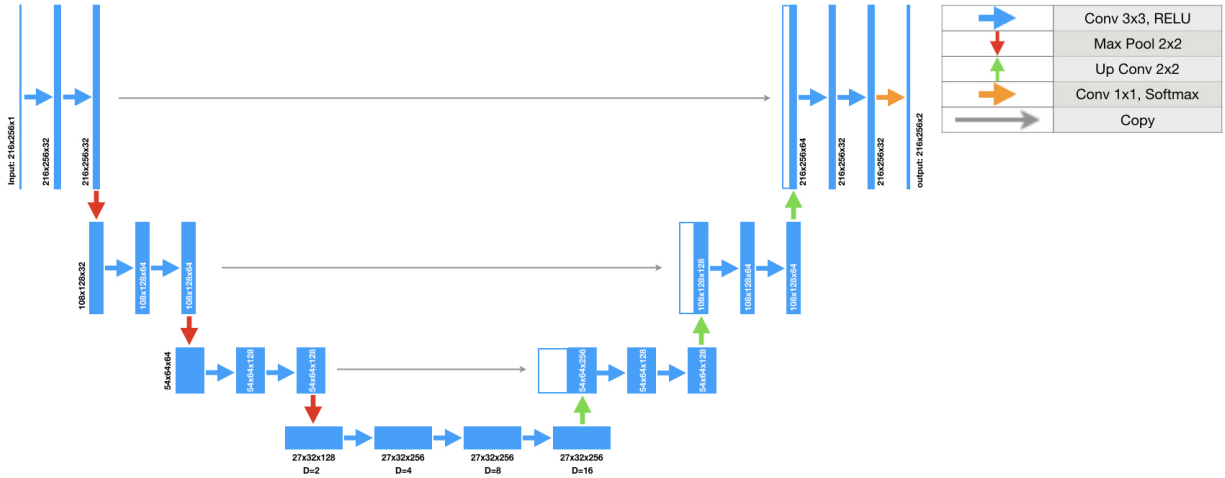


Figure 7: Architecture of the U-Net model with dilated convolution layers after downsampling. D corresponds to the dilation factor of that layer.

Figure 8: An example of the dilated U-Net correctly predicting only one right ventricle from the same image as above.

After training, we created contours for both testing sets using the regular and dilated U-Net models. The text files were sent to the competition moderators for evaluation. We report the final results in Section 4 below.

4 Results

4.1 Model Evaluation and Validation

In this section, we will discuss the results for the two on the testing datasets. The first model was trained for 500 epochs with no dropout and a learning rate of 0.001. The second model evaluated was trained for 500 epochs with no dropout, a learning rate of 0.001, and dilated convolution layers. The accuracy and loss plots that were generated during training of both models can be seen in Figure 9. The learning curves for both models follow a standard shape, with loss values converging. The validation loss for the regular U-Net seems to be slightly greater than the training loss, indicating overfitting. For the dilated U-Net, the validation loss and training loss are almost equal, with a smaller difference between the two compared to the regular U-Net. This conjecture based on the learning curves is supported by the average training and validation Dice scores reported in Section 3.3 and Table 1.

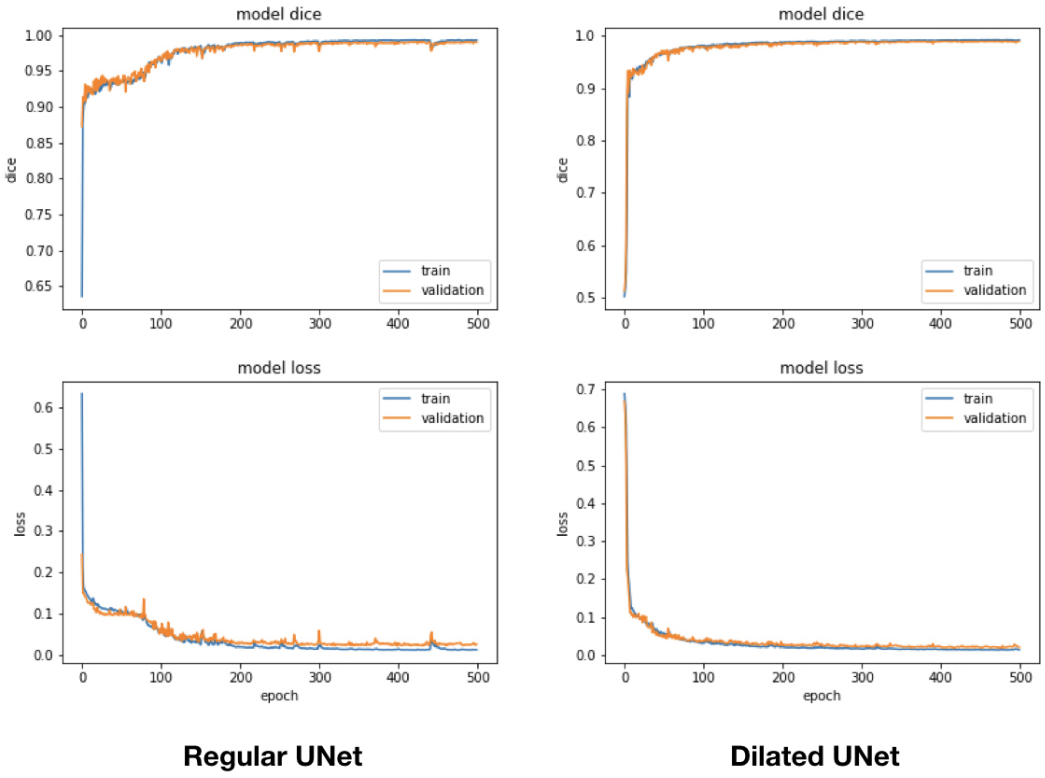


Figure 9: The accuracy and loss plots for both the regular and dilated U-Net models. Both models converge towards a minimum loss, but the regular U-Net seems to be overfitting more than the dilated U-Net as the difference between the training and validation loss for the regular U-Net appear to have a greater difference than the respective losses for the dilated U-Net.

The regular U-Net model achieved an average Dice score of 0.75 with a standard deviation of 0.31 when predicting endocardium contours across both test sets. For epicardium predictions, the regular U-Net scored an average Dice of 0.78 with a standard deviation of 0.28. Our model with dilated convolutional layers received an average Dice score of 0.80 with a standard deviation of 0.26 on endocardium contours. It received an average Dice of 0.86 with a standard deviation of 0.18 for epicardium contours. The results compared to the state of the art benchmark can be seen in the Table 1.

Model	Training	Validation	Test
Endocardium Results			
FCN - benchmark	-	-	0.84(0.21)
U-Net	0.90(0.11)	0.76(0.28)	0.75(0.31)
U-Net w/dilation	0.90(0.11)	0.80(0.25)	0.80(0.26)
Epicardium Results			
FCN - benchmark	-	-	0.86(0.20)
U-Net	0.93(0.08)	0.78(0.29)	0.78(0.28)
U-Net w/dilation	0.93(0.07)	0.86(0.17)	0.86(0.18)

Table 1: This table shows the average Dice scores (with standard deviation in parentheses) for each model on the training, validation and testing datasets.

4.2 Justification

Compared to the benchmark model (Fully-convolutional Network), our models do not perform better. At best, the models match the benchmark performance. In generating the endocardium contours, the benchmark achieved an average Dice score of 0.84 compared to the average Dice scores of 0.75 and 0.80 for our regular and dilated U-Net models, respectively. Clearly there are still some improvements that need to be made to the models to improve their endocardium performance. For epicardium predictions, the benchmark scored an average Dice of 0.86 which was matched by the dilated U-Net model. The regular U-Net model scored an average Dice of 0.78 for epicardium predictions. In general, the models seemed to score higher when predicting epicardium contours than endocardium contours. In addition to the state-of-the-art benchmark model, our models also perform worse than manual human performance. The average Dice score for a manual contour created by an expert is 0.90 with a standard deviation of 0.10. Neither of the models we developed beat that Dice score. Thus, we cannot currently use the models we created as a replacement for manual segmentation. A more useful solution would be to generate the automatic contours, and then have the expert improve the finer details manually. This would use the best of both methods. It would save time by performing an initial automatic segmentation, but it would also retain the accuracy of a manual contour by having an human check it's performance and make smaller edits to the model output.

5 Conclusion

5.1 Free-Form Visualization

Adding the dilated convolution layers to the U-Net architecture helped with reducing the number of multiple contours that were incorrectly detected in an image. However, the model was still making mistakes. The dilated U-Net performed much better on images where the right ventricle was large, corresponding to the end diastole stage of the cardiac cycle. When predicting on an image where the right ventricle was at its smallest size (end systole) then even our dilated U-Net model struggled to make any prediction at all. Figure 10 shows an example where the model performs well when the right ventricle area is large, while it predicts no right ventricle in an image where the actual ventricle is very small in area. Improving the model performance on end systole images when the right ventricle is small in area is an important step to take.

5.2 Reflection

In this project, we developed a neural network model that would automatically segment the right ventricle from cardiac MRI images. We worked on all parts of the machine learning development phase, from data loading and preprocessing to model implementation, training, and refinement. We were also responsible for generating contours on the test data sets. Since we were working with a small data set, our data preprocessing involved normalizing the images and augmenting them to generate more examples for our model during training. In order to accomplish this, we made use of the common `ImageDataGenerator` object available in the Keras API. For model development, we used the Keras API to add layers to our model and select specific hyper-parameters. After training, we implemented classes to generate predictions on unseen data.

(a)

(b)

Figure 10: Examples of (a) end diastole and (b) end systole images where the model makes a strong prediction for one and no prediction for the other.

There were many challenging aspects in solving this problem, which also made it quite interesting. By working on this project, we were able to learn how real machine learning occurs in real world systems and the constraints that are sometimes hidden in a controlled project. For example, we were working with real MRI images, and the properties of these images were not uniform. Some images were rotated 90 degrees, and this was something we had to be mindful of when preprocessing our data. In addition, the challenge of working with image segmentation of the right ventricle was the size of the right ventricle relative to the actual image. The majority of the pixels in our input images were background pixels, and so we had to do more preprocessing to normalize the images in order to allow the model to gain insight on a greater variety of pixel intensities when training. We also had a small number of labeled images in our training set, and had to augment the images to reduce overfitting. This problem came about because labeling images for this particular problem set is time consuming. The small dataset obstacle is not a unique issue, and is a good example of what one can expect when working on machine learning problems in the real world. More than anything, working on this capstone project has exposed us to real data and forced us to come up with solutions to preprocess that data to then eventually feed into our model for successful training. This is indicative of the actual work involved in machine learning projects, where a lot of time is spent gathering and cleaning data before models can even begin to be implemented. These challenging aspects of the project turned out to be one of the greatest lessons learned about machine learning in the real world.

Another challenge faced in this project was understanding the issues that our model had with predicting multiple right ventricles in a single image. It made sense as to why this problem occurred for our model, because the deepest layers did not have a context of the whole image when setting weights. In order to solve this problem, we had to do more research about techniques used in image segmentation, specifically what is done when each pixel is being classified and an output map is being created based on the image. This led us to the topic of dilated convolution layers. After researching dilated convolutions, and finding they had characteristics that would be a useful to solve our issue, we had to learn how to implement them in the Keras API. In the end, we saw an improvement in the performance of the neural network on the image data. Again, working to find a solution to an obstacle led to a deeper understanding of neural networks and deep learning. This is a deeper understanding only developed after working with more complex CNNs.

5.3 Improvement

Even though we made good progress on solving many challenges, there are many improvements we can still make. The first improvement is to increase the accuracy of the model on the testing set. Right now, it does not beat the benchmark model on endocardium images, and it does not come close to gold standard manual contour is created by an expert. Accomplishing this task would involve experimenting with different numbers of layers or possibly entirely different architectures for the network. We were using the U-Net architecture

as it is commonly used for segmentation problems where the output has to be at the same size as the input image. However, there are other architectures that we could implement and evaluate to determine which one achieves the highest accuracy.

Another improvement that could be made is to increase the generality of the model. This would mean creating a model that can segment all the sections of the heart in a single MRI image. In order to introduce this generality to our model, we would have to use a new architecture and also gather new data. Since this new problem would be a multi-class segmentation problem, where we would have more than two possible classes for each pixel in the segmentation map, meaning a modified U-Net model would not be sufficient. Instead we could implement a SegNet architecture to perform this multi-class segmentation. The SegNet architecture is similar in structure to the U-Net, where there is a downsampling block followed by an upsampling block to create an output map of various segmented classes. The main difference between U-Net and SegNet lies in the way information from the downsampling layers is copied over to the upsampling block. In U-Net, we transpose copies of the layers before max-pooling and concatenate these copies with the corresponding upsampling steps to double the feature size. This allows each upsampling step to learn back any relevant features that were lost due to pooling. In SegNet, instead of creating copies of the downsampled layers, the network retains the location of the max-indices that contributed to the max-pooling during downsampling [6]. This allows for the creation of a segmentation map that performs better on images that have multiple segmentation regions to classify. We would also need to work on gathering cardiac MRI image data that has multiple regions contoured or labeled. This would have to be a completely new dataset, as currently, we can find datasets for left and right ventricle segmentation separately, but having labels of the same regions marked for a single image would be essential for training this combination neural network.

All in all, these are just two ideas of how to improve the model we developed for this specific problem domain. There are other improvements that could be made to make this model more useful in a medical setting. Integration with current medical software would definitely be a long term use of this model. It is exciting to learn about the uses and applications this project can have in improving the ability for a physician to better diagnose diseases in a patient. This problem domain of using machine learning to improve a physicians insight on patient care is an important area to explore for the future. The problem we worked on is just one of many that can be used to improve the quality of care people receive in hospitals and other medical settings. The medical field is definitely ripe with machine learning problems that can be explored in the future.

References

[1] C. Petitjean, M. A. Zuluaga, and W. Bai, "Right ventricle segmentation from cardiac mri: A collation study," *Medical Image Analysis*, vol. 19, no. 1, pp. 187–202, 2015.

[2] P. V. Tran, "A fully convolutional neural network for cardiac segmentation in short-axis mri." arXiv:1604.00494, 2017.

[3] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation." arXiv:1505.04597, 2015.

[4] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation." arXiv:1411.4038, 2015.

[5] B. Kayalıbay, G. Jensen, and P. van der Smagt, "Cnn-based segmentation of medical imaging data." arXiv:1701.03056, 2017.

[6] M. Shah, "Dilated convolutions and kronecker factored convolutions." `https://meetshah1995.github.io/semantic-segmentation/deep-learning/pytorch/visdom/2017/06/01/semantic-segmentation-over-the-years.html`. Accessed: 2018-06-30.

[7] F. Yu and V. Koltun, "Multi-scale context aggregation by dilated convolutions." arXiv:1511.07122, 2016.

[8] L. Y. Sulimowicz, "Semantic segmentation using fully convolutional networks over the years." `https://medium.com/@lisulimowicz/dilated-convolutions-and-kronecker-factored-convolutions-b42ed58b2bc7`. Accessed: 2018-06-30.