Semantic Segmentation

pab734 wxw870 zxw780 txd844 cxb980 gxk636

1 Introduction

We have split our datasets in training, validation and testing data. The dataset consists of CMR images and the masks for the training images. Our training contains only images,

We will be designing a neural network to segment the CMR images into four distinct regions as accurately and reliably as possible. The masks produced will simplify the analysis of the data greatly.

We want our neural network to be able to appropriately segment the test data to the four regions to a high accuracy. We want our network to avoid underfitting and overfitting, instead producing a reasonable mask of any given image. In order for us to achieve this we will be focusing on finding the optimum network architecture and loss function for image segmentation. Furthermore, by using optimisers and augmenting the dataset we aim to increase the accuracy and reliability of the segmentation.

2 Implementation

2.1 Data Augmentation

We augmented the data by using numpy to flip our data set (fliplr() and flipud()) and rotated the data anti-clockwise in steps of 90 degrees (rot90()) three times to increase the size of the dataset from 100 images to 600. This increase in variety in our data will decrease the risk of the network remembering the training data and help prevent a large difference in segmentation performance from training to validation in the network.

2.2 FCN

The idea of FCN is based on CNN. It replaced the fully connected layer of CNN by fully convolutional layer and it can classify images at the pixel level. Therefore, it solved the semantic segmentation of image segmentation. Finally, the loss of SoftMax classification is calculated pixel by pixel, which is equivalent to one training label for each pixel. Overall, FCN replaces the fully connected layer at the end of CNN with the convolutional layer, and the output an image which is already labeled.

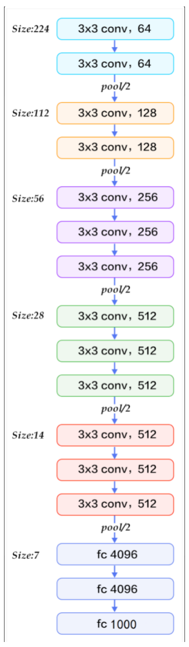


Figure 1: VGG16 Structure

In our FCN, we first use the VGG16 convolutional part as FCN convolutional part. In original VGG16, the image input size is 224*224*3 and the max channels are 512. For each convolutional layer, only the number of channels is changed, and the size is become one half after each maxpool layer, which means after 5 convolutional layers, the size becomes 1/8 of its original size. However, this structure might not work well on this dataset, as the magnetic resonance (MR) dataset is 96*96*1. It is necessary to adapt the layers and parameters to fit our dataset. We keep each convolutional layer parameter as same as VGG16 except the input and output channel. The input channel for first layer which is input layer is obviously 1 and the output channel becomes 16. The output channel of rest four convolutional layers is 32, 64, 128, 256 respectively. That's all of the convolutional part. An advantage of this structure is when doing convolutional part to extract the features, the size never changed and it means information that is not lost at the edge of the image. And the maxpool layer reduce the size of feature map and improve the anti-interference ability of network.

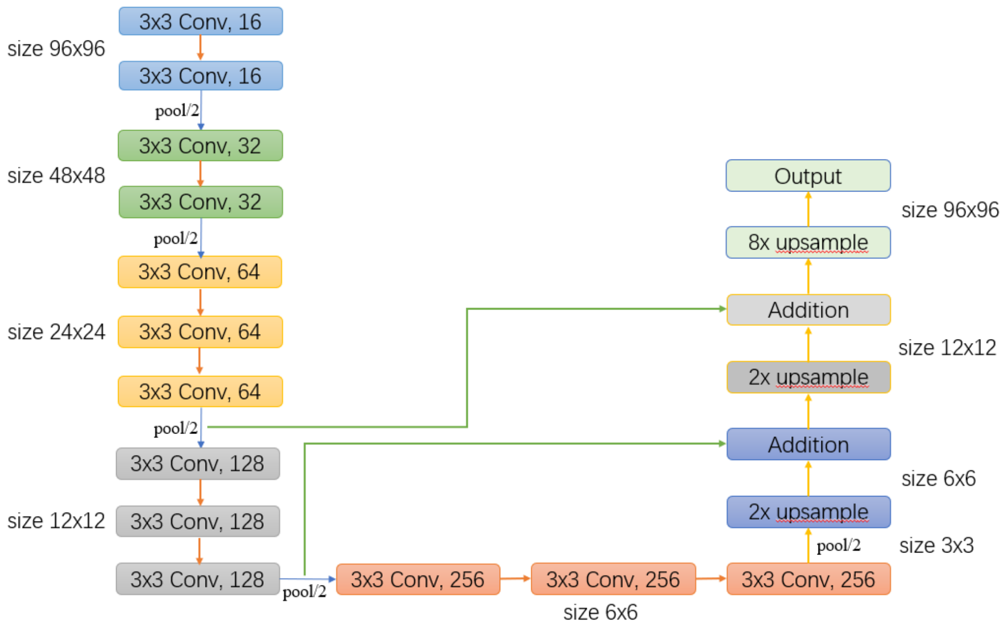


Figure 2: VGG16-FCN Structure

Then the upsample part aims to make the convolved image return to the input size. The easiest way to implement this is to use upsampling at a factor of 32. However, as we can see on figure2 the predicted mask, it is generally similar with origin mask, the details are very poor. In addition, the accuracy of validation in this case is only around 70%.

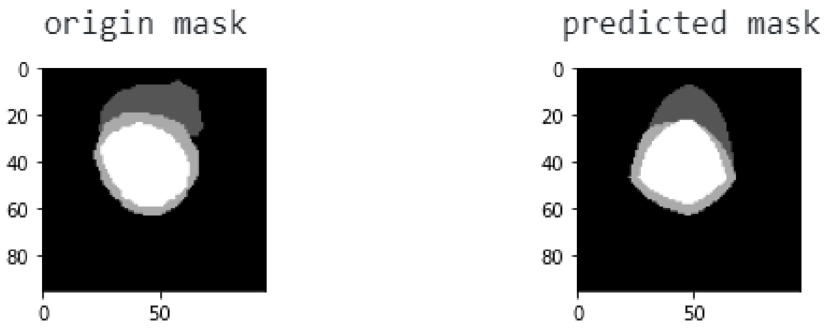


Figure 3: Upsampling as factor of 32

Therefore, we use upsampling at a factor of 2 first, apply this to the output of convolutional layer, and add with the output of fourth convolutional part, we get "x1". Using "x1" and add with the output of third convolutional part, we get "x2". Finally, using "x2" with upsampling at a factor of 8, we get the predicted output. Now, the predicted mask as shown on figure 3, most of the details are fine, and we can reach the maximum validation accuracy about 88%.

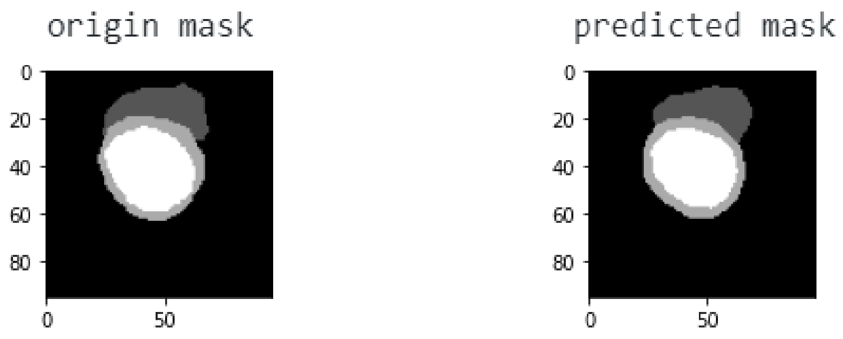
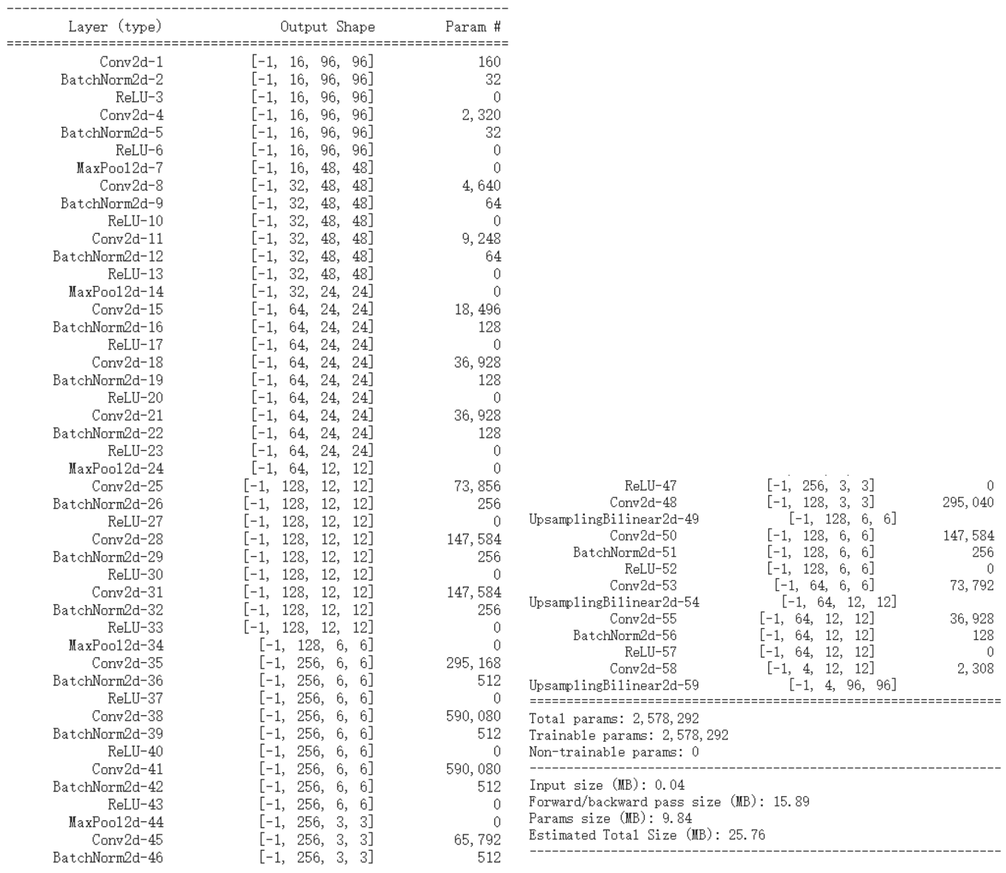


Figure 4: Upsampling as factor of 2,2,8

In summary, there are 5 convolutional part and 3 upsample part in our VGG16-FCN network. The output shape of each convolutional part after maxpool are 48*48*16, 24*24*32, 12*12*64, 6*6*128, 3*3*256. Total parameters in this network are 2,578,292 and total storage is about 25.76MB.



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 16, 96, 96]             160
       BatchNorm2d-2           [-1, 16, 96, 96]              32
              ReLU-3           [-1, 16, 96, 96]               0
            Conv2d-4           [-1, 16, 96, 96]           2,320
       BatchNorm2d-5           [-1, 16, 96, 96]              32
              ReLU-6           [-1, 16, 96, 96]               0
         MaxPool2d-7           [-1, 16, 48, 48]               0
            Conv2d-8           [-1, 32, 48, 48]           4,640
       BatchNorm2d-9           [-1, 32, 48, 48]              64
             ReLU-10           [-1, 32, 48, 48]               0
           Conv2d-11           [-1, 32, 48, 48]           9,248
      BatchNorm2d-12           [-1, 32, 48, 48]              64
             ReLU-13           [-1, 32, 48, 48]               0
        MaxPool2d-14           [-1, 32, 24, 24]               0
           Conv2d-15           [-1, 64, 24, 24]          18,496
      BatchNorm2d-16           [-1, 64, 24, 24]             128
             ReLU-17           [-1, 64, 24, 24]               0
           Conv2d-18           [-1, 64, 24, 24]          36,928
      BatchNorm2d-19           [-1, 64, 24, 24]             128
             ReLU-20           [-1, 64, 24, 24]               0
           Conv2d-21           [-1, 64, 24, 24]          36,928
      BatchNorm2d-22           [-1, 64, 24, 24]             128
             ReLU-23           [-1, 64, 24, 24]               0
        MaxPool2d-24           [-1, 64, 12, 12]               0
           Conv2d-25          [-1, 128, 12, 12]          73,856
      BatchNorm2d-26          [-1, 128, 12, 12]             256
             ReLU-27          [-1, 128, 12, 12]               0
           Conv2d-28          [-1, 128, 12, 12]         147,584
      BatchNorm2d-29          [-1, 128, 12, 12]             256
             ReLU-30          [-1, 128, 12, 12]               0
           Conv2d-31          [-1, 128, 12, 12]         147,584
      BatchNorm2d-32          [-1, 128, 12, 12]             256
             ReLU-33          [-1, 128, 12, 12]               0
        MaxPool2d-34            [-1, 128, 6, 6]               0
           Conv2d-35            [-1, 256, 6, 6]         295,168
      BatchNorm2d-36            [-1, 256, 6, 6]             512
             ReLU-37            [-1, 256, 6, 6]               0
           Conv2d-38            [-1, 256, 6, 6]         590,080
      BatchNorm2d-39            [-1, 256, 6, 6]             512
             ReLU-40            [-1, 256, 6, 6]               0
           Conv2d-41            [-1, 256, 6, 6]         590,080
      BatchNorm2d-42            [-1, 256, 6, 6]             512
             ReLU-43            [-1, 256, 6, 6]               0
        MaxPool2d-44            [-1, 256, 3, 3]               0
           Conv2d-45            [-1, 256, 3, 3]          65,792
      BatchNorm2d-46            [-1, 256, 3, 3]             512
             ReLU-47            [-1, 256, 3, 3]               0
           Conv2d-48            [-1, 128, 3, 3]         295,040
UpsamplingBilinear2d-49            [-1, 128, 6, 6]               0
           Conv2d-50            [-1, 128, 6, 6]         147,584
      BatchNorm2d-51            [-1, 128, 6, 6]             256
             ReLU-52            [-1, 128, 6, 6]               0
           Conv2d-53             [-1, 64, 6, 6]          73,792
UpsamplingBilinear2d-54           [-1, 64, 12, 12]               0
           Conv2d-55           [-1, 64, 12, 12]          36,928
      BatchNorm2d-56           [-1, 64, 12, 12]             128
             ReLU-57           [-1, 64, 12, 12]               0
           Conv2d-58            [-1, 4, 12, 12]           2,308
UpsamplingBilinear2d-59            [-1, 4, 96, 96]               0
================================================================
Total params: 2,578,292
Trainable params: 2,578,292
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.04
Forward/backward pass size (MB): 15.89
Params size (MB): 9.84
Estimated Total Size (MB): 25.76
----------------------------------------------------------------
```

Figure 5: VGG16-FCN Structure Summary

2.3 DeepLab v3 plus

Besides the network we implement with FCN and VGG-16, we also explore many other network architectures. The Deeplab series is the most interesting to us, and we have chosen the latest one to perform semantic segmentation on our task.
The state-of-art network Deeplab v3 plus was proposed in 2018 by Google and has been open-sourced on TensorFlow, which combines the advantages of various networks of recent years.

As seen in the image above, the network uses the Encoder-Decoder framework which was used before such as network U-Net. The encoder part gradually reduces spatial dimensionality, while the decoder gradually restores spatial dimensionality by using bilinear upsampling with low-level features to detail the information.

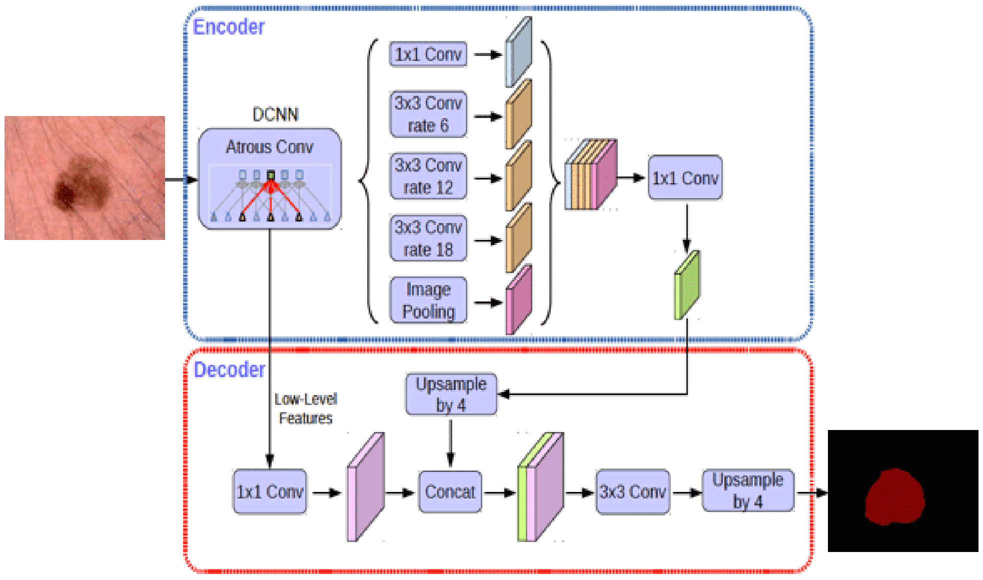


Figure 6: Deeplab v3 plus network

For the encoder part, DCNN (Deep Convolutional Neural Network) is the first module. Deeplab v3 plus using a new network as the backbone for features map extraction called Aligned Xception. The main difference of the Aligned Xception from the original Xception is that it replaced all the max-pooling section inside, using depthwise separatable convolution and atrous convolution instead. Atrous convolution expands the filter field of view to add a broader background, which resolves the loss of spatial information caused by max-pooling in the previous version and increase the output resolution. Also, the number of repetitions of Middle flow has been doubled to 16 to achieve better performance.

After the DCNN part is the ASPP (Atrous Spatial Pyramid Pooling) module. This module is designed to address the fact that the same object can have different sizes in the picture. To improve the model's ability to recognise different sizes of the same object, a common method was Spatial Pyramid Pooling, where the original image is resized to different scales, fed into the same network to obtain different feature maps, and then concatenate it, which can indeed improve the accuracy. But another problem is that it is too slow. DeepLab v2 solved this problem by introducing the ASPP (Atrous Spatial Pyramid Pooling), the feature maps are convolved in parallel with different spatial pyramid pooling layers with atrous convolution and the output is concatenated to obtain the segmentation result of the image. Deeplab v3 plus maintains this design.

For the decoder part, the decoder first reduces channels of low-level feature map with a 1x1 filter convolutional layer, then concatenates the output of the encoder which was after bilinear 4x upsampling, then use two 3x3 filters convolutional layer and one 1x1 filter convolutional layer before the final bilinear 4x upsampling, and then upsampling to get the result.

After we figure out the network structures, we try to train the model with our data set. We didn't make any changes to the network initially, and we were able to train the model with our own training set to achieve 95% accuracy, but the validation set only achieved 62-70% accuracy. This is a sign of overfitting.

So, we did the same thing we did for VGG-16, by changing the channels inside of the network, in addtion, we add dropout inside to avoid overfitting. We keep the output stride to be 16 rather than 32, which means the output feature map of encoder will be the size of 6*6 (height and width reduced by 16 times compared to the original image size 96*96).
The best result then we get from the validation set is 88%, but it only achieves 85% on Kaggle.
In summary, the DeepLab series has been one of the most important semantic segmentation networks to date. It started with an atrous convolution and gradually added more features from other networks, such as SPP and Decoder, to make it even stronger.

2.4 Loss functions

We first considered 3 loss functions: Cross Entropy Loss, Negative log-likelihood loss, and Soft dice loss. Cross entropy and Negative log-likelihood already have implementations in Torch.nn (CrossEntropyLoss() and

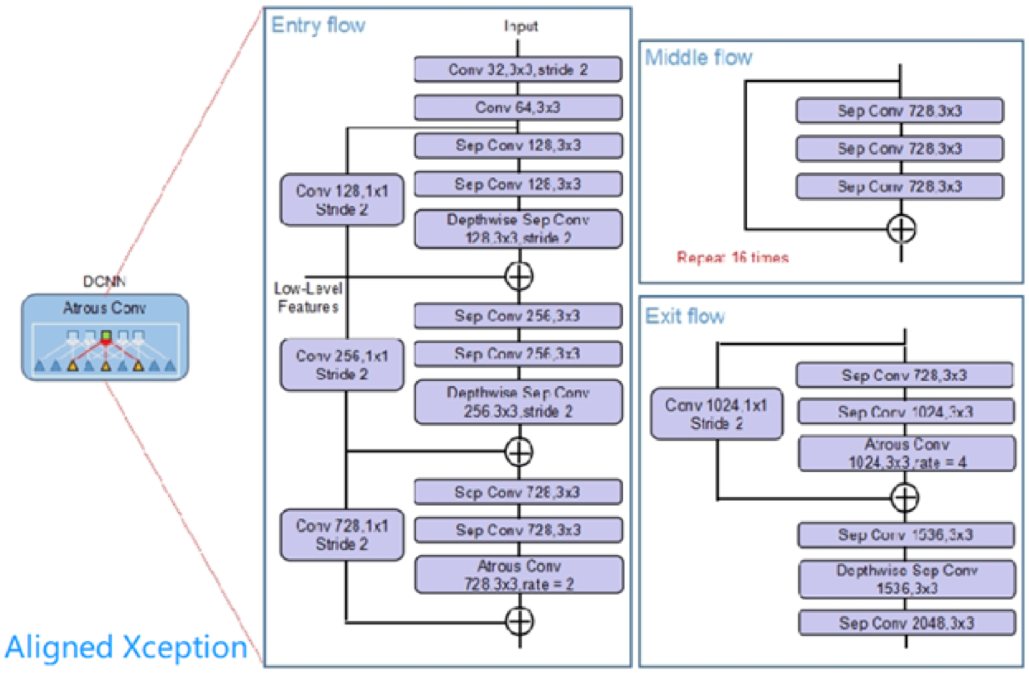


Figure 7: The structure of modified Aligned Xception.

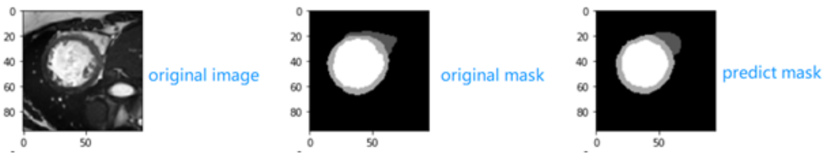


Figure 8: Predict mask example

NLLLoss2d()) so we used those. However, we had to implement Soft Dice loss ourselves. The class follows the general structure of a Torch.nn module, with the only argument being an optional smoothing factor. The forward function implements the process of finding soft dice loss. The shape of the input is found and appropriate axes created, and then the true positives, false positives and false negatives are returned by the helper function getresults(). These are used to calculate the Dice Coefficient (dc), which is negated and returned as the loss value. The source code for SoftDiceLoss() is shown below:

```
class SoftDiceLoss(nn.Module):
    def __init__(self,smooth=0.1):
        super(SoftDiceLoss, self).__init__()
        self.smooth = smooth

    def forward(self, x, y, loss_mask=None):
        shp_x = x.shape
        axes = list(range(2, len(shp_x)))

        true_p, false_p, false_n = get_results(x, y, axes)
        dc = (2 * true_p + self.smooth) / (2 * true_p + false_p + false_n + self.smooth)
        dc = dc.mean()

        return 1-dc
```

However, individual loss functions have weaknesses; these weaknesses can be bridged by considering several loss functions in conjunction. The first such function is “CrossDice”, a basic combination of Soft dice loss and Cross entropy loss. The forward function for this model is shown below. The individual loss values are calculated, before they are multiplied to produce an overall loss value.

```
def forward(self, inputs, targets):
        cross_loss = self.cross.forward(inputs, targets)
        dice_loss = self.dice.forward(inputs, targets)
```

```
    comb_loss = cross_loss*dice_loss

    return comb_loss
```

On the original network, CrossDice performed well while training for the first 20 epochs but dropped off massively afterwards. I think this is down to learning rate and a bad optimiser, I'll have to check it out

3 Experiment

3.1 Loss Functions

The first consideration regarding There are few parameters to be varied in loss functions. In Cross Entropy loss, the only parameter to be varied is applying weights to the different classes to lessen the impact of an unbalanced dataset. The default weight of a class is 1, such that a weight of 0.5 will halve the impact of the given class. In this task there are 4 classes representing different areas of the mask, and intuitively the background is much more significant than the rest of the image. To test this hypothesis, we first ran 4 test configurations where one weight was set to 0.5 and the rest left as 1. The model was only trained for 20 epochs on a subset of the data for efficiency. The results of this test are shown in the table below

Weighted classes vs Validation accuracy

X	None	Class 1	Class 2	Class 3	Class 4
Accuracy	0.61644	0.72185	0.67240	0.67338	0.64554

As can be seen above, a lesser weighting for Class 1 produces far better results then the other tests performed.

4 Conclusion

We have found that After experimenting with different architectures we found that the DeepLab V3 Plus model was the most appropriate for our task. We used VGG-16 network architecture by changing the channels in the network. By adding dropout we could reduce the amount of overfitting present in the network. We found that by adding optimisers we could improve the accuracy of the network. We added Adam as the optimiser algorithm for our network as it is effective at increasing the accuracy of our predictions. The most optimum loss function was CrossDice() as it is a combination of Soft dice loss and Cross entropy loss which counteract each others weaknesses and creates a more effective loss function.
By finding the optimum features of our network we are able to achieve ??? percent accuracy on training and validation and can appropriatly predict the segmentation on the given images.