Coursework 2021/2022

Semantic Image Segmentation

Neural Computation

Group 18

2058651
mxb1138@student.bham.ac.uk

2169399
zxw099@student.bham.ac.uk

2048464
yxf965@student.bham.ac.uk

2069692
ycn992@student.bham.ac.uk

2070865
xxy869@student.bham.ac.uk

2080521
jxc1259@student.bham.ac.uk

2170099
kxw099@student.bham.ac.uk

University of Birmingham, *29/11/2021*

0 Setup

0.0 Downloader

Please install and import the google drive downloader by running the following cell.

In [ ]:

```
!pip install googledrivedownloader
from google_drive_downloader import GoogleDriveDownloader as gdd
```

0.1 Data

Just run the following cell to download and extract the data, best models and their configs temporarily for the current session.

In [ ]:

```
# Download the prepared data
gdd.download_file_from_google_drive(file_id="1dpq_OnbLvm22wh4P8qWAO_S_bYthLrcj",
                                    dest_path="./extra.zip",
                                    unzip=True)

# Remove the not needed anymore zip file
!rm -rf extra.zip
```

```
Downloading 1DN1FtmJINiymDeIoelITC7bIg2C21CxL into ./data.zip... Done.
Unzipping...Done.
```

0.3 Packages

The project depends on the following additional packages.

In [ ]:

```
!pip install numpy matplotlib torch torchvision opencv-python albumentations monai einops timm t
qdm
```

0.4 Libraries

Import all the necessary libraries as shown below:

In [ ]:

```
# Import the main libraries
import os
import json
import math
import numpy as np
import torch
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F
# import torch.utils.data as data
import torch.utils.model_zoo as model_zoo
import cv2
import albumentations as A
import matplotlib.pyplot as plt

# Import specific components
from tqdm.notebook import tqdm
from glob import glob
from monai.losses import *
from monai.networks import one_hot
from torch.utils.data import DataLoader

# External files
from segnet import SegNet
from fcn import FCN8s, VGGNet
from unet import UNet
from effunet import *
```

1 Introduction

1.1 Overview

This report describes the process of completing the continuous coursework assessment on semantic image segmentation as well as provides code snippets for reproducing the results. With the chosen architecture approach, the final performance on the public test set is 86.1%.

1.2 Task Description

This year's coursework task focuses on semantic medical image segmentation of cardiovascular magnetic resonance (CMR) pictures. In particular, the segmented masks consist of 4 regions: the background region (black), the left ventricle (LV) region (white), the myocardium (Myo) region (white gray) and the right ventricle (RV) region (dark gray). We are provided with a prepared dataset from ACDC challenge [1] with 200 images. The rest of the notebook describes the strategies and the research done in order to maximize the segmentation accuracy based on the Dice loss function [2].

1.3 Relevant Work

Numerous algorithms have been developed for segmentation tasks, ranging from the earliest methods, such as watersheds [3], region-growing [4], to the current state of art self-supervised and transformer approaches, such as Microsoft's BEiT [5] or Swin Transformer [6]. With the advent of deep-learning, the most popular choices for image segmentation have also shifted in recent years [7]. Models like FCN [8], U-Net [14], SegNet [9] and DeepLab [10] all have shown incredible performance in accuracy and although they are a bit behind compared to the most modern solutions, they still serve as a baseline for future approaches. For example, one of the current state of art networks for segmentation is (still CNN-based) EfficientNet + NAS-FPN [11]. Considering the ACDC dataset itself, recent research has focused on GAN based methods [12, 13] which provide competitive results. Our approach tries to combine several ideas and justifies final choices based on the experiments. Some model considerations, such as vision-transformer-based, are, however, omitted due to model complexity, lack of computation power and insufficient data for thorough training.

1.4 Teamwork and Workflow

The main platforms used for collaboration were GitHub, Google Drive and Google Colaboratory. The code was structured, documented and integrated in a git repository and the relevant parts were moved to this notebook. Papers, plans and other code was contained in a shared folder on Google Drive. The communication platforms were Slack and Zoom. The team has divided the tasks for each member to work on individually, where each member would work on a part of the code and a part of the report. The produced code and report parts were reviewed by other members before submission.

2 Implementation

2.1 Network Architecture

There were several models proposed by the team members. With references to official repositories and research papers, we reimplemented them in our own project and selected the ones that have shown the best performance. Some models, however had to be discarded, such as transformers due to their complexity, resource limitation and unstable performance. Some external experiments have been done and the results will be shown in the report.

Candidate Architectures

FCN

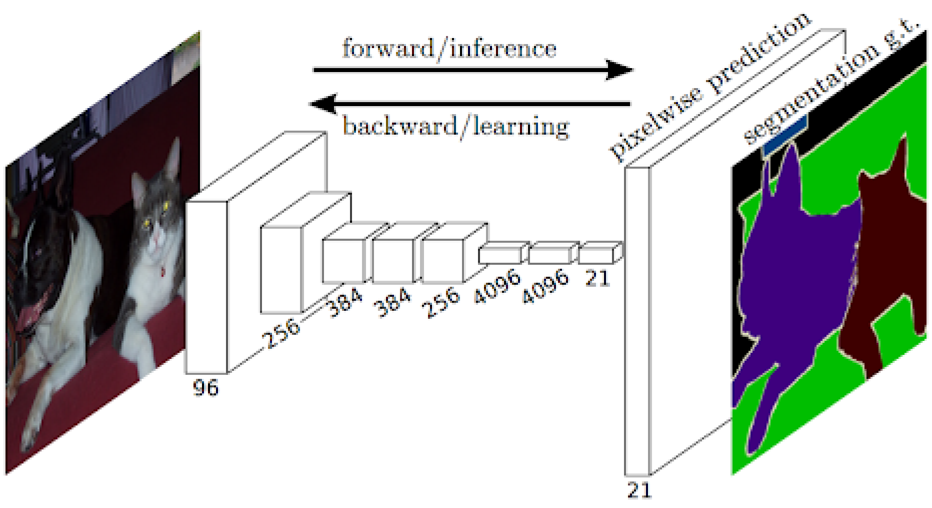


Figure 1. FCN Architecture[8]

Long et al. [8] proposed a milestone architecture, Fully Convolutional Network (FCN), in 2015. It utilises fully convolutional layers combined with up-sample layers to solve the semantic segmentation task.

Different from the classic convolutional neural network (CNN) which uses the fully connected layer to obtain fixed-length feature vectors for classification tasks, FCN can accept input images of any size. The deconvolution layer is also employed to up-sample the feature map of the last convolutional layer in order to restore it to the same size of the input image. As a consequence, a prediction can be generated for each pixel, while preserving the spatial information in the original input image. A pixel-level classification is conducted on the up-sampled feature map. Furthermore, a skip architecture is also introduced to take advantage of the feature spectrum that combines deep, coarse, semantic information and shallow, fine, appearance information.

FCN Family

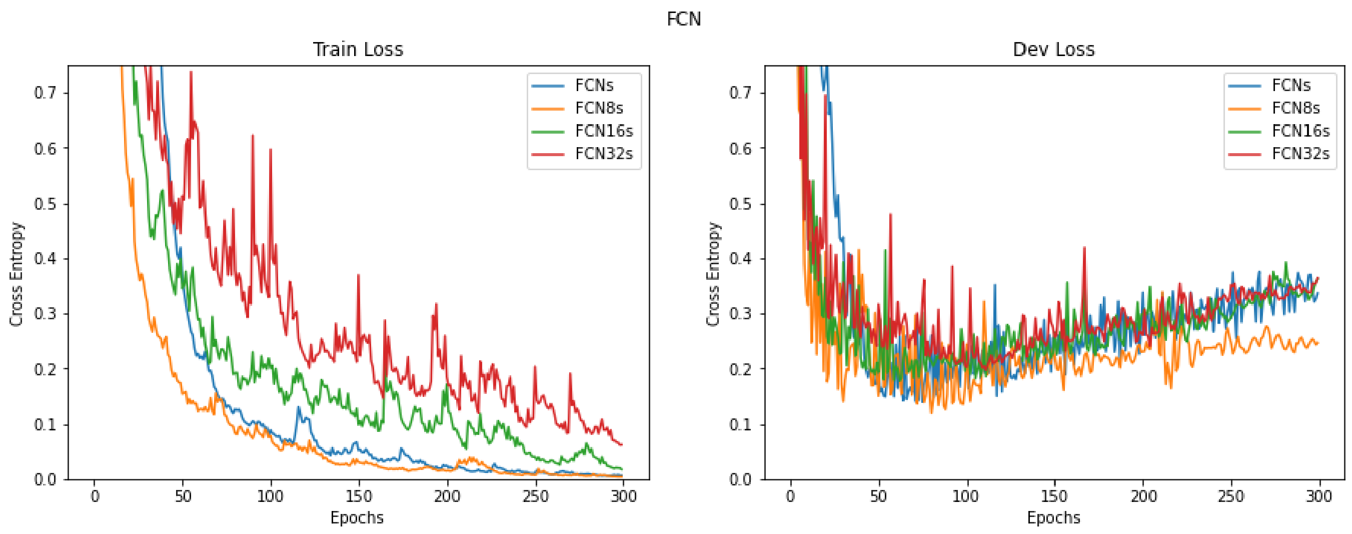


Figure 2. FCN Families

UNet

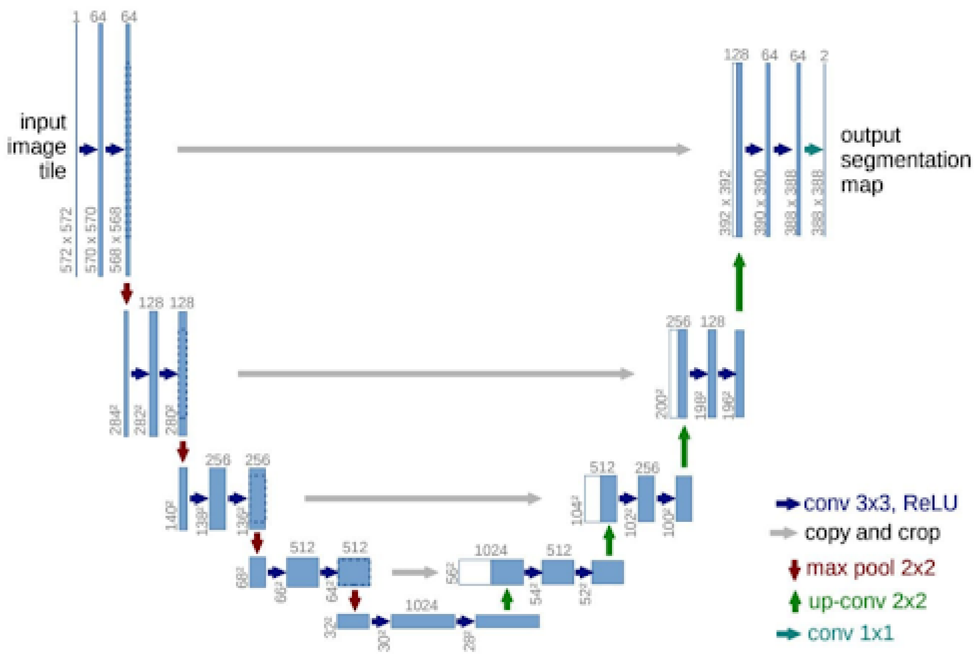


Figure 3. U-Net Architecture[14]

U-Net was proposed by Ronneberger et al. in [14]. With U-Net and data augmentation, their solution can effectively learn from few-annotated images. Built upon FCN, U-Net further modifies up-sampling part of FCN, resulting in a U-shaped architecture which allows the network to propagate context information to higher resolution layers. The skip-connections are also different where “concatenation” is used, while “add” is used in FCN.

Apart from architecture, mirror-padding is also proposed in the overlap-tile strategy to allow seamless segmentations for arbitrary large images. Since zero padding may bring distortion and noise as the number of convolutional layers increases, mirror-padding is proposed to tackle the issue.

For this model, many contraction blocks are used to make up the contraction section. An input is given to each block, then two 3x3 convolution layers followed by a 2x2 max pooling layer are applied to the output from the block. This would double the number of kernels/ feature maps and allow the architecture to successfully learn complex structures. The layer of the bottom of the ‘U’ enables smooth transition from the contraction layer to the expansion layer, where two 3x3 convolution neural networks (CNNs) are first applied, succeeded by a 2x2 up-convolution layer.

Due to its prevalence and its rewarding modification based on FCN, U-Net was selected as one of the candidate models.

Depth of U-Net

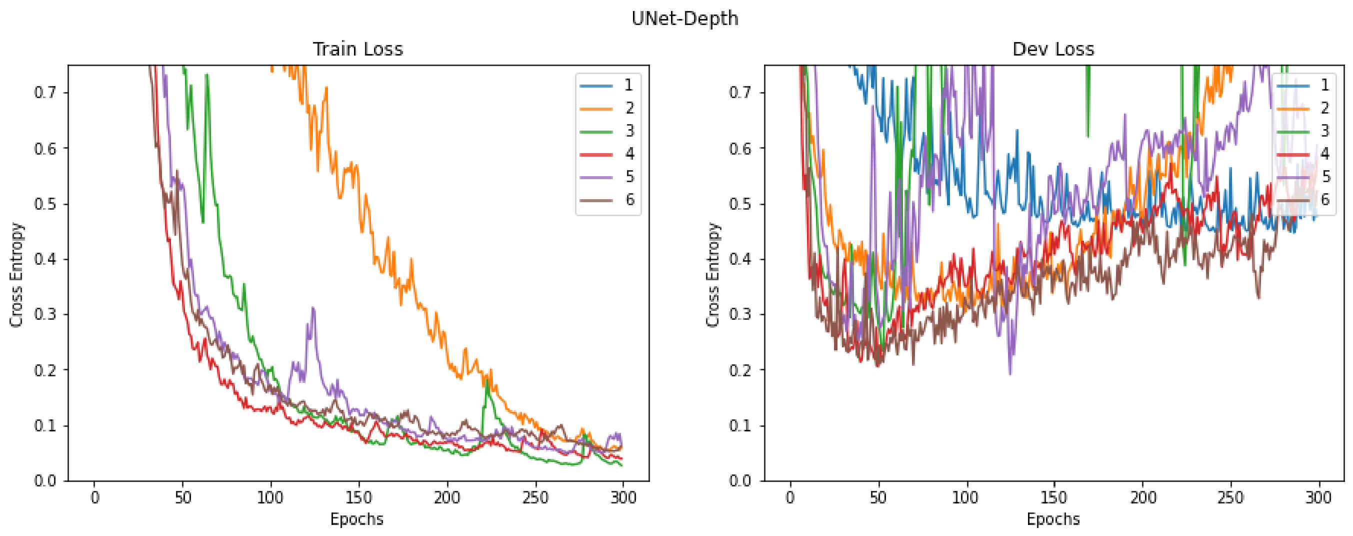


Figure 4. Training Loss and Validation Loss for Different Depths of U-Net plotted against the number of Epochs

The above figures indicate how the selection of depth for the model affects the loss, which is calculated by the Cross Entropy Loss function. The training loss is shown on the left and the validation loss is shown on the right, and depths 1 - 6 are plotted in different colors. It can be seen that both depth 4 (red line) and depth 6 (brown line) show the best performance from depth 1 to depth 6. Given that the training time of depth 6 is twice as much as depth 4, the U-Net with depth of 4 is chosen for the candidate model.

However, as there are candidate models that achieved a better performance, this architecture is not chosen as the final model. Code related to building the U-Net model can be found in the Appendix.

Final Candidates

1. DeepLab

DeepLab series are milestone architectures in semantic segmentation. DeepLab v1 was proposed by Chen et al.. in [15]. Based on Deep Convolutional Neural Network (DCNN), DeepLab v1 further utilised hole algorithm (dilated convolution). The Conditional Random Field (CRF) is also incorporated to accurate localisation. Due to its great performance, it is the model that is preserved to the end.

DeepLab v2 [16] proposed Atrous Spatial Pyramid Pooling (ASPP) inspired by Spatial Pyramid Pooling (SPP). At the stage before pixel classification, ASPP is inserted to explits multi-scale features by employing multiple parallel filters with different rates. ASPP with different atrous rates can effectively capture multi-scale information. However, as the atrous rate increases, the 3x3 filter degrades to 1x1 filter as only the center weight is effective. DeepLab v3 [17] further modified ASPP with global pooling and 1x1 conv layer in order to solve the above issue.

In DeepLab v3+ [18], the DCNN part is regarded as encoder and the up-sampling part is referred to as decoder in order to form an Encoder-Decoder structure which has been proven to be an efficient architecture in segmentation tasks. Due to the effectiveness of atrous convolution and Encoder-Decoder architecture, we eventually selected DeepLab v3+ as the main architecture for hyper-parameter tuning.

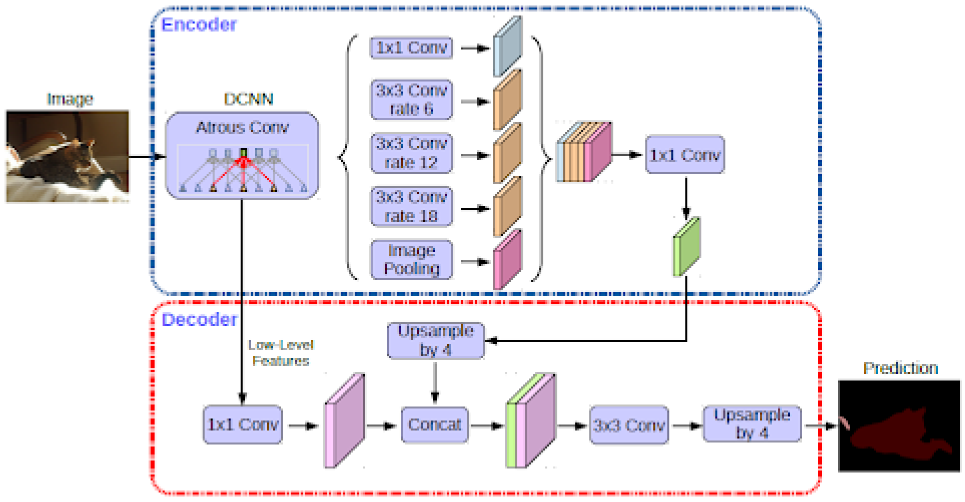


Figure 5. DeeplabV3+ Architecture [18]

DeeplabV3+ with Different Backbones

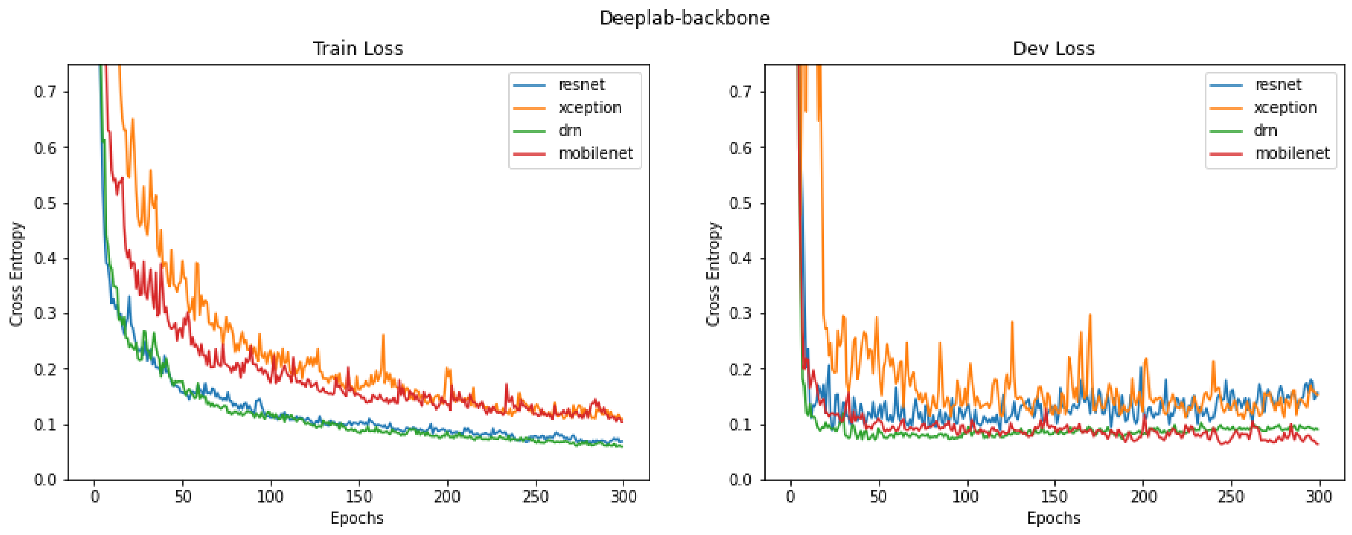


Figure 6. DeeplabV3+ Backbones [18]

Dilated Residual Networks (DRN) was chosen as the backbone for the DeepLabV3+ architecture. The above figure also shows the effect of 0 - 300 epochs, but this time it is showing the difference in different backbone used for the DeepLabV3+ architecture. Although deeplab-mobilenet has the lowest validation loss, it converges at a lower speed and has high training error. At the same time, DRN and ResNet have similar training loss and validation loss. However, ResNet shows unstable behaviour during the validation process where the line fluctuated a lot, where in comparison, DRN has a much more stable performance.

Dilated Residual Network

The Dilated Residual Network(DRN) [19] is chosen as the backbone, where bigger feature map sizes were kept compared to other backbones while having better mechanisms to alleviate degradation when the number of layers is increasing. Dilated Residual Network (DRN) introduces dilated convolution into ResNet. It contains two kinds of blocks. And all the convolution blocks contain (a normal convolution-batch normalisation-ReLU). The first one(blue) is a normal convolution block.

The second one is the Residual Block(Red), which is made up of 3 convolution blocks.

- The initial convolution block will reformat the channels into the intended number.
- The second convolution block uses custom stride, dilated factor and padding.
- The last convolution block will expand the features to 4*intended_channels.
- Finally there is a residual path connected between input and the third block. If $c_i \neq c$, the residual be downsampled by a 1x1 convolution block with the custom stride. The convolution block here has excluded the ReLU activation function.

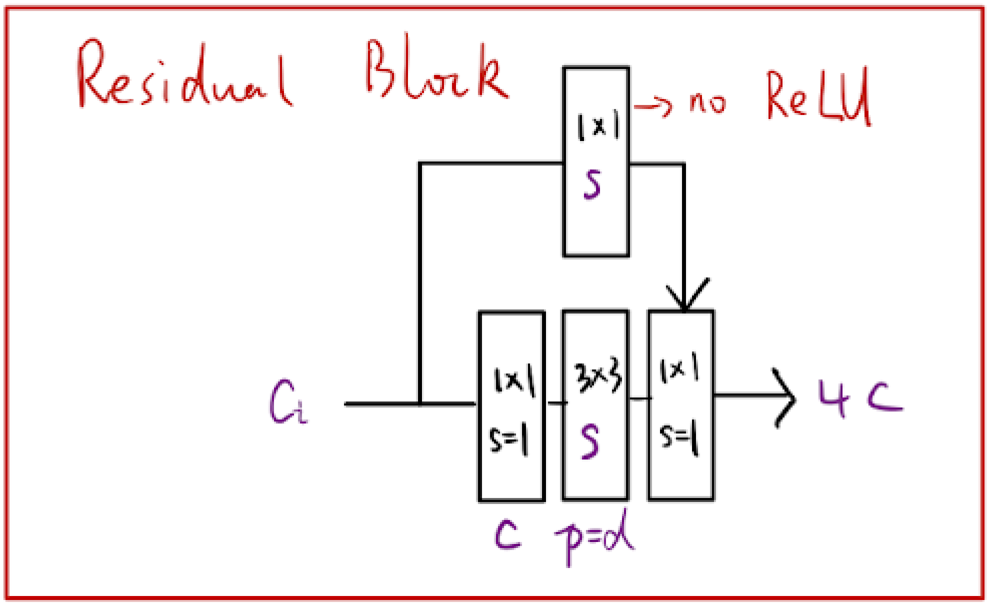


Figure 7. DRN Residual Block

Backbone Architecture

Figure 8. (see below) is a visualisation of our backbone architecture – DRN-d54. It has 9 layers in total.

- Layers 0-2 each contain only 1 normal convolution block.
- Layer 3 uses 3 residual blocks. The first residual block has stride = 2 and it has the same effect as a pooling layer, which reduces the input height and width by half. It is designed to replace max pooling layers, because the author found that this is an effective way to remove the gridding artifacts. While the rest blocks only use stride = 1.
- Layer 4 is similar to layer 3, except that it has 1 more residual block.
- Layer 5 and layer 6 uses dilated convolution, namely d=2 and d=4. They do not reduce the input height and width while increasing the receptive field. And the dilated convolution is designed to replace the downsampling layers in the ResNet.
- Layers 7-8 each contain only 1 normal convolution block. They are designed for removing the gridding artifacts.

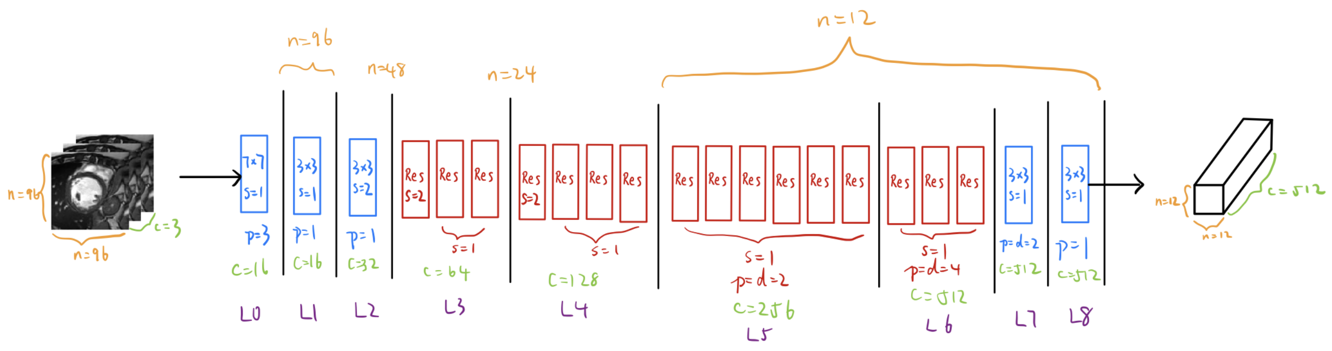


Figure 8. DRN-d54 Architecture

DeepLabV3+

DeepLabV3+ is an encoder-decoder architecture. Given an input, the encoder will generate 2 outputs, namely high level features and low level features. These 2 outputs will be fed into the decoder and the decoder will generate the prediction.

Encoder

For encoder, as Figure 9. below shows:

- The output of layer 3 was chosen to be the low level features. Note that residual blocks will expand the channels by 4 times, so the output has 64x4-256 channels.
- The output of the DRN network will go through the 'Spatial pyramid pooling module'. After going through the dilated convolution blocks with d = [1, 12, 24, 36] and average pooling blocks. We get 5 different features maps which have the same size as the original DRN output. These 5 features maps will be concatenated and then the channels become 256x5 = 1280. After a simple 1x1 convolution block, the channel reduces to 256. And this is the high level features.

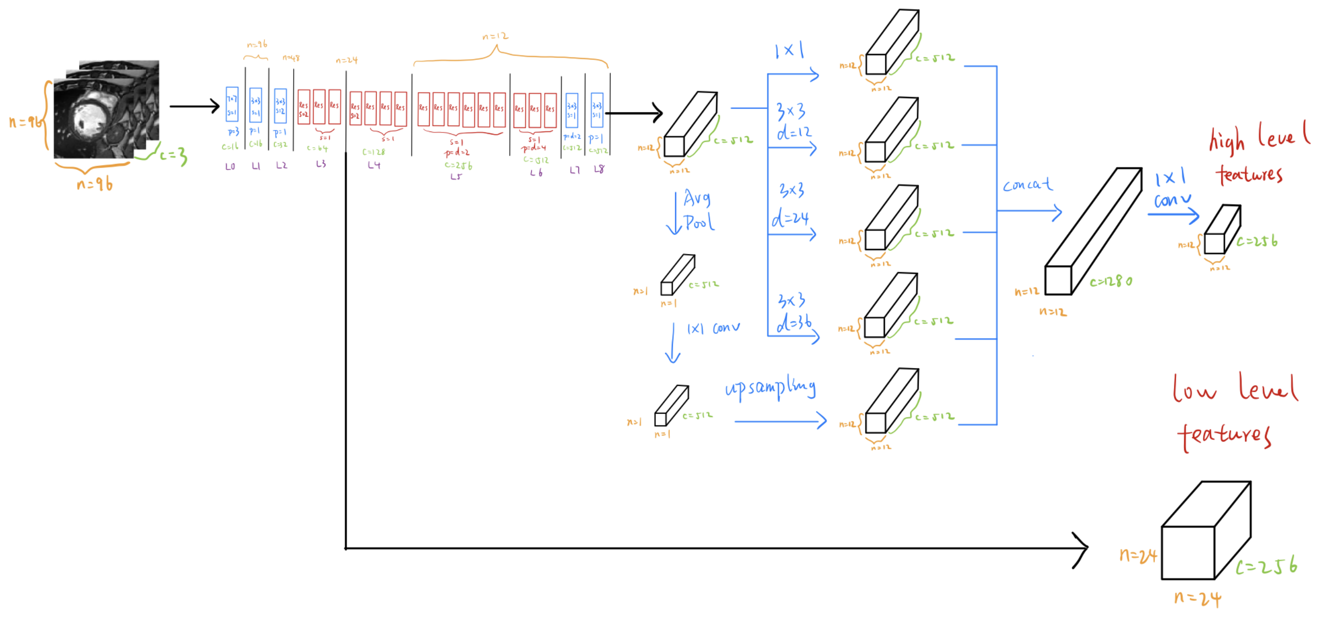


Figure 9. DeepLabV3+ Encoder Architecture

Decoder

For the decoder, as shown below in Figure 10: the high level features will be upsampled in order to align with the low level features size. The low level features will go through a 1x1 convolution block and the channels are reduced to 48. After a concatenation of the processed features, the feature maps will go through 2 3x3 convolution blocks, and the channels are reduced to 256. Then, a 1x1 convolution will reduce the channels to 4. The output was then upsampled in order to align with the ground truth segmentation mask. Finally, we take the maximum index in each channel as the predicted class.

The architecture illustrated below is the final chosen architecture as it achieved the best performance and accuracy, where the implementation is shown in the next section.

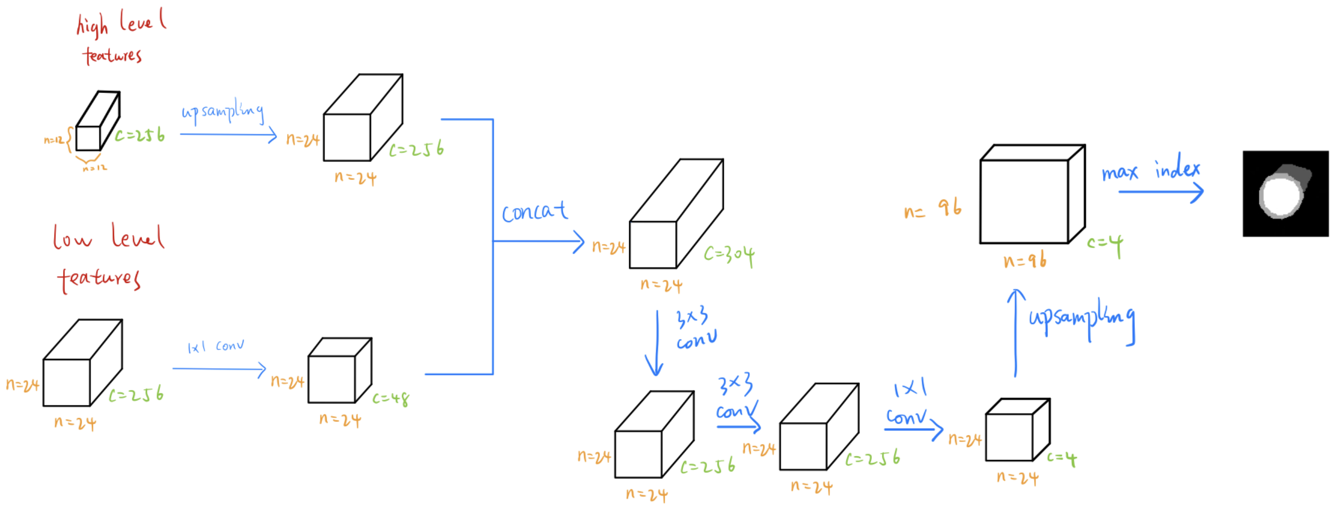


Figure 10. DeepLabV3+ Decoder Architecture

Code Implementation

The following code is based on: [https://github.com/jfzhang95/pytorch-deeplab-xception (https://github.com/jfzhang95/pytorch-deeplab-xception)](https://github.com/jfzhang95/pytorch-deeplab-xception) with adaptation to fit the task that is given. The implemented architecture is exactly the same as the self-drawn illustrations above so explanations of architecture are not written within the code to avoid redundancy.

Dilated Residual Block

In [ ]:

```
#@markdown Dilated Residual Block
class DrnResidualBlock(nn.Module):

    # the block will always output a feature map with 4*out_channels channels
    expansion = 4

    def __init__(self, input_channels, output_channels, stride=1, downsample=None, dilation=(1,
1), BatchNorm=None):
        """init DrnResidualBlock

        Args:
            input_channels: the input channel size of the feature map
            output_channels: the output channel size of the feature map, they will be 4 times bi
gger than this
            stride: stride for the second convolution block
            downsample: a convolution block for formatting the residual size
            dilation: atrous rate
            BatchNorm : BatchNorm class for neural network
        """
        super(DrnResidualBlock, self).__init__()
        # 1x1 conv block, to  reformat the input_channels to output_channels
        self.conv1 = nn.Conv2d(input_channels, output_channels, kernel_size=1, bias=False)
        self.bn1 = BatchNorm(output_channels)
        # 3x3 conv block, where dilated convolution happens
        self.conv2 = nn.Conv2d(output_channels, output_channels, kernel_size=3, stride=stride,
                               padding=dilation[1], bias=False,
                               dilation=dilation[1])
        self.bn2 = BatchNorm(output_channels)
        # 1x1 conv block,  expand the features to 4*output_channels
        self.conv3 = nn.Conv2d(output_channels, output_channels * 4, kernel_size=1, bias=False)
        self.bn3 = BatchNorm(output_channels * 4)
        self.relu = nn.ReLU(inplace=True)
        # If input_channels != output_channels, the residual be downsampled
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        residual = x

        # 1x1 conv block, to  reformat the input_channels to output_channels
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        # 3x3 conv block, where dilated convolution happens
        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        # 1x1 conv block,  expand the features to 4*output_channels
        out = self.conv3(out)
        out = self.bn3(out)

        # If input_channels != output_channels, the residual be downsampled
        if self.downsample is not None:
            residual = self.downsample(x)

        # residual connection
        out += residual
        out = self.relu(out)

        return out
```

DRN

In [ ]:

```
#@markdown Dilated Residual Network
class DRN(nn.Module):

    def __init__(self, res_block_cls, layer_block_size,
                 channels=(16, 32, 64, 128, 256, 512, 512, 512),
                 BatchNorm=None):
        """init DRN

        Args:
            res_block_cls: the residual block class
            layer_block_size: a list specifying how many blocks each layer have
            stride: stride for the second convolution block
            downsample: a convolution block for formatting the residual size
            dilation: atrous rate
            BatchNorm : BatchNorm class for neural network
        """
        super(DRN, self).__init__()
        self.input_channels = channels[0]
        self.out_dim = channels[-1]

        # 7x7 conv, well padded so that the output size is same as input size
        self.layer0 = nn.Sequential(
            nn.Conv2d(3, channels[0], kernel_size=7, stride=1, padding=3,
                      bias=False),
            BatchNorm(channels[0]),
            nn.ReLU(inplace=True)
        )

        # 3x3 conv, well padded so that the output size is same as input size
        self.layer1 = self.build_conv_layer(channels[0], layer_block_size[0], stride=1, BatchNor
m=BatchNorm)
        # 3x3 conv, stride = 2, the input size is input size/2 now
        self.layer2 = self.build_conv_layer(channels[1], layer_block_size[1], stride=2, BatchNor
m=BatchNorm)

        # 4 residual layers
        # input size will be halved
        self.layer3 = self.build_residual_layer(res_block_cls, channels[2], layer_block_size[2],
stride=2, BatchNorm=BatchNorm)
        # input size will be halved
        self.layer4 = self.build_residual_layer(res_block_cls, channels[3], layer_block_size[3],
stride=2, BatchNorm=BatchNorm)
        # dilated convolution
        self.layer5 = self.build_residual_layer(res_block_cls, channels[4], layer_block_size[4],
dilation=2, new_level=False, BatchNorm=BatchNorm)
        # dilated convolution
        self.layer6 = self.build_residual_layer(res_block_cls, channels[5], layer_block_size[5],
dilation=4, new_level=False, BatchNorm=BatchNorm)

        # 3x3 conv dilated convolution, for removing gridding artifacts
        self.layer7 = self.build_conv_layer(channels[6], layer_block_size[6], dilation=2, BatchN
orm=BatchNorm)
        # 3x3 conv, for removing gridding artifacts
        self.layer8 = self.build_conv_layer(channels[7], layer_block_size[7], dilation=1, BatchN
orm=BatchNorm)

        self._init_weight()

    def _init_weight(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_(0, math.sqrt(2. / n))
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()


    def build_residual_layer(self, res_block_cls, ouput_channels, block_num, stride=1, dilation=
1,
                             new_level=True, BatchNorm=None):
        assert dilation == 1 or dilation % 2 == 0
        # downsample is designed for the residuals block internally
        downsample = None
        if stride != 1 or self.input_channels != ouput_channels * res_block_cls.expansion:
            downsample = nn.Sequential(
                nn.Conv2d(self.input_channels, ouput_channels * res_block_cls.expansion,
                          kernel_size=1, stride=stride, bias=False),
                BatchNorm(ouput_channels * res_block_cls.expansion),
            )

        layers = list()
        # the first block is with custom stride, this is designed to replace max pooling layers
 for removing gridding artifacts
        layers.append(res_block_cls(
            self.input_channels, ouput_channels, stride, downsample,
            dilation=(1, 1) if dilation == 1 else (
                dilation // 2 if new_level else dilation, dilation), BatchNorm=BatchNorm))
        # the ouput channel size of the residual block is always input_channels*4
        self.input_channels = ouput_channels * res_block_cls.expansion
        # build the rest residual blocks
        for i in range(1, block_num):
            layers.append(res_block_cls(self.input_channels, ouput_channels, dilation=(dilation,
dilation), BatchNorm=BatchNorm))

        return nn.Sequential(*layers)

    def build_conv_layer(self, output_channels, block_num, stride=1, dilation=1, BatchNorm=None
):
        modules = []
        # build several conv-NB-ReLU Blocks, the first block will always have stride = 1
        for i in range(block_num):
            modules.extend([
                nn.Conv2d(self.input_channels, output_channels, kernel_size=3,
                          stride=stride if i == 0 else 1,
                          padding=dilation, bias=False, dilation=dilation),
                BatchNorm(output_channels),
                nn.ReLU(inplace=True)])
            self.input_channels = output_channels
        return nn.Sequential(*modules)

    def forward(self, x):
        # conv block
        x = self.layer0(x)
        x = self.layer1(x)
        x = self.layer2(x)

        # residual block
        x = self.layer3(x)
        low_level_feat = x
        x = self.layer4(x)
        x = self.layer5(x)
        x = self.layer6(x)

        # conv block
        x = self.layer7(x)
        x = self.layer8(x)

        return x, low_level_feat

def build_drn_d_54(BatchNorm, pretrained=True):
    """Builds a drn_d_54 backbone

    Args:
        BatchNorm : BatchNorm class for neural network
        pretrained: use pretrained model

    Returns:
        model (nn.Module): A model built on the provided specifications
    """
    model = DRN(DrnResidualBlock, [1, 1, 3, 4, 6, 3, 1, 1], BatchNorm=BatchNorm)
    if pretrained:
        pretrained = model_zoo.load_url('http://dl.yf.io/drn/drn_d_54-0e0534ff.pth')
        del pretrained['fc.weight']
        del pretrained['fc.bias']
        model.load_state_dict(pretrained)
    return model
```

Encoder - Spatial Pyramid Pooling

In [ ]:

```
#@markdown Spatial Pyramid Pooling
# ASPP---------------------------------------------------------------------------------------
---
class ASPPModule(nn.Module):
    def __init__(self, input_channels, output_channels, kernel_size, padding, dilation, BatchNor
m):
        """init ASPPModule

        Args:
            input_channels: the input channel size of the feature map
            output_channels: the output channel size of the feature map
            kernel_size: the convolution kernel size
            padding: feature map padding, usually uses 'same size' padding
            dilation: atrous rate
            BatchNorm : BatchNorm class for neural network
        """
        super(ASPPModule, self).__init__()
        self.atrous_conv = nn.Conv2d(input_channels, output_channels, kernel_size=kernel_size,
                                     stride=1, padding=padding, dilation=dilation, bias=False)
        self.bn = BatchNorm(output_channels)
        self.relu = nn.ReLU()

        self._init_weight()

    def forward(self, x):
        x = self.atrous_conv(x)
        x = self.bn(x)

        return self.relu(x)

    def _init_weight(self):
        """init the decoder weights for deeplabv3+
        use kaiming normal distribution to init kernel weights
        use 1 for BatchNorm weights and 0 for BatchNorm bias
        """
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                torch.nn.init.kaiming_normal_(m.weight)
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()

class SpatialPyramidPooling(nn.Module):
    def __init__(self, backbone, output_stride, BatchNorm):
        """init Spatial Pyramid Pooling

        Args:
            backbone (string): one of 'resnet','mobilenet','drn' or 'xception'
            output_stride: Output stride explains the ratio of the input image size to the outpu
t feature map size. For drn, it's 96/12 = 8
            BatchNorm : BatchNorm class for neural network
        """
        super(SpatialPyramidPooling, self).__init__()
        if backbone == 'drn':
            input_channels = 512
        elif backbone == 'mobilenet':
            input_channels = 320
        else:
            input_channels = 2048
        if output_stride == 16:
            dilations = [1, 6, 12, 18]
        elif output_stride == 8:
            dilations = [1, 12, 24, 36]
        else:
            raise NotImplementedError

        # 1x1 conv
        self.aspp1 = ASPPModule(input_channels, 256, 1, padding=0, dilation=dilations[0], BatchN
orm=BatchNorm)
        # 3x3 dilated conv for drn
        self.aspp2 = ASPPModule(input_channels, 256, 3, padding=dilations[1], dilation=dilations
[1], BatchNorm=BatchNorm)
        # 3x3 dilated conv for drn
        self.aspp3 = ASPPModule(input_channels, 256, 3, padding=dilations[2], dilation=dilations
[2], BatchNorm=BatchNorm)
        # 3x3 dilated conv for drn
        self.aspp4 = ASPPModule(input_channels, 256, 3, padding=dilations[3], dilation=dilations
[3], BatchNorm=BatchNorm)
        # average pooling layer, will do upsampling to match other aspp size
        self.global_avg_pool = nn.Sequential(nn.AdaptiveAvgPool2d((1, 1)),
                                             nn.Conv2d(input_channels, 256, 1, stride=1, bias=Fa
lse),
                                             BatchNorm(256),
                                             nn.ReLU())
        # 1x1 conv, reduce the concatenated channels to 256
        self.conv1 = nn.Conv2d(1280, 256, 1, bias=False)
        self.bn1 = BatchNorm(256)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.5)
        self._init_weight()

    def forward(self, x):
        # we have 5 aspp modules, and they will all be concatenated
        x1 = self.aspp1(x)
        x2 = self.aspp2(x)
        x3 = self.aspp3(x)
        x4 = self.aspp4(x)
        # do upsampling to match other aspp size
        x5 = self.global_avg_pool(x)
        x5 = F.interpolate(x5, size=x4.size()[2:], mode='bilinear', align_corners=True)
        x = torch.cat((x1, x2, x3, x4, x5), dim=1)

        # 1x1 conv, reduce the concatenated channels to 256
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)

        return self.dropout(x)

    def _init_weight(self):
        """init the decoder weights for deeplabv3+
        use kaiming normal distribution to init kernel weights
        use 1 for BatchNorm weights and 0 for BatchNorm bias
        """
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                # n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                # m.weight.data.normal_(0, math.sqrt(2. / n))
                torch.nn.init.kaiming_normal_(m.weight)
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()


def build_spatial_pyramid_pooling(backbone, output_stride, BatchNorm):
    """Builds spatial pyramid pooling block

    Args:
        backbone (string): one of 'resnet','mobilenet','drn' or 'xception'
        output_stride: the output stride of backbone
        BatchNorm : BatchNorm class for neural network

    Returns:
        model (nn.Module): A deeplabv3+ decoder
    """
    return SpatialPyramidPooling(backbone, output_stride, BatchNorm)
```

Decoder

In [ ]:

```
#@markdown Decoder
# Decoder-----------------------------------------------------------------------------------------
-------
class Decoder(nn.Module):
    def __init__(self, num_classes, backbone, BatchNorm):
        """init a decoder for deeplabv3+

        Args:
            num_classes: the classes in prediction
            backbone (string): one of 'resnet','mobilenet','drn' or 'xception'
            BatchNorm : BatchNorm class for neural network

        Returns:
            model (nn.Module): A deeplabv3+ decoder
        """
        super(Decoder, self).__init__()
        if backbone == 'resnet' or backbone == 'drn':
            low_level_input_channels = 256
        elif backbone == 'xception':
            low_level_input_channels = 128
        elif backbone == 'mobilenet':
            low_level_input_channels = 24
        else:
            raise NotImplementedError

        # for low level features, this convolution block reduce the channels to 48
        self.conv1 = nn.Conv2d(low_level_input_channels, 48, 1, bias=False)
        self.bn1 = BatchNorm(48)
        self.relu = nn.ReLU()

        # the convolution blocks for the concatenated feature maps, the output of this block is
 going to be upsampling
        self.last_conv = nn.Sequential(nn.Conv2d(304, 256, kernel_size=3, stride=1, padding=1, b
ias=False),
                                       BatchNorm(256),
                                       nn.ReLU(),
                                       nn.Dropout(0.5),
                                       nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, b
ias=False),
                                       BatchNorm(256),
                                       nn.ReLU(),
                                       nn.Dropout(0.1),
                                       nn.Conv2d(256, num_classes, kernel_size=1, stride=1))
        self._init_weight()


    def forward(self, x, low_level_feat):
        # the channels of low level features will be reduced to 48
        low_level_feat = self.conv1(low_level_feat)
        low_level_feat = self.bn1(low_level_feat)
        low_level_feat = self.relu(low_level_feat)

        # high level feature will be upsampled to algin with low level feature size
        x = F.interpolate(x, size=low_level_feat.size()[2:], mode='bilinear', align_corners=True
)
        x = torch.cat((x, low_level_feat), dim=1)
        # get a 4 channel tensor, to be upsampled
        x = self.last_conv(x)
        return x

    def _init_weight(self):
        """init the decoder weights for deeplabv3+
        use kaiming normal distribution to init kernel weights
        use 1 for BatchNorm weights and 0 for BatchNorm bias
        """
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                torch.nn.init.kaiming_normal_(m.weight)
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()

def build_decoder(num_classes, backbone, BatchNorm):
    """Builds a decoder

    Args:
        num_classes: the classes in prediction
        backbone (string): one of 'resnet','mobilenet','drn' or 'xception'
        BatchNorm : BatchNorm class for neural network

    Returns:
        model (nn.Module): A deeplabv3+ decoder
    """
    return Decoder(num_classes, backbone, BatchNorm)
```

Integration

In [ ]:

```
#@markdown Integration
# Model-----------------------------------------------------------------------------------------
----
class DeepLab(nn.Module):
    def __init__(self, backbone='drn', output_stride=8, num_classes=4):
        """init deeplabv3+

        Args:
            backbone (string): one of 'resnet','mobilenet','drn' or 'xception'
            output_stride: Output stride explains the ratio of the input image size to the outpu
t feature map size. For drn, it's 96/12 = 8
            num_classes: the classes in prediction
        """
        super(DeepLab, self).__init__()
        if backbone == 'drn':
            output_stride = 8

        BatchNorm = nn.BatchNorm2d

        self.backbone = build_backbone(backbone, output_stride, BatchNorm)
        self.aspp = build_spatial_pyramid_pooling(backbone, output_stride, BatchNorm)
        self.decoder = build_decoder(num_classes, backbone, BatchNorm)

    def forward(self, input):
        # get high level features and low level features
        x, low_level_feat = self.backbone(input)
        # high level features will go to aspp for further extraction
        x = self.aspp(x)
        # feed aspp output and low level features into the decoder
        x = self.decoder(x, low_level_feat)
        # do upsampling for the predicted mask to match the input size
        x = F.interpolate(x, size=input.size()[2:], mode='bilinear', align_corners=True)

        return x

# backbone--------------------------------------------------------------------------------------
-------
def build_backbone(backbone, output_stride, BatchNorm):
    """Builds spatial pyramid pooling block

    Args:
        backbone (string): one of 'resnet','mobilenet','drn' or 'xception'
        output_stride: the output stride of backbone
        BatchNorm : BatchNorm class for neural network

    Returns:
        model (nn.Module): A deeplabv3+ decoder
    """
    if backbone == 'resnet':
        return ResNet101(output_stride, BatchNorm)
    elif backbone == 'xception':
        return AlignedXception(output_stride, BatchNorm)
    elif backbone == 'drn':
        return build_drn_d_54(BatchNorm)
    elif backbone == 'mobilenet':
        return MobileNetV2(output_stride, BatchNorm)
    else:
        raise NotImplementedError
```

2. Ensemble

Another one of the final architecture choices is the ensemble model which averages the performance of 5 famous segmentation models (elaborated in later sections). Each model provides raw outputs for the segmentation image (4 channels for different segments). The outputs were passed through softmax function channel-wise to get the respective segment probabilities and those probabilities were simply averaged.

Code implementation

In [ ]:

```
class VotingSegmenter(nn.Module):
    """Majority-based voting ensamble.

    This model ensamble decides on the final outcome by averaging the
    results of 5 different models.
    """

    def __init__(self, ms, cs):
        """Initializes the voting ensemble.

        Args:
            ms (list(nn.Module)): The list of segmentation models
            cs (list(int)): The list of input channels for each model
        """
        super().__init__()

        # Initialize models and channels
        self.ms = nn.Sequential(*ms)
        self.cs = cs


    def forward(self, x):
        """Performs forward pass of the model ensemble.

        Args:
            x (torch.tensor): The raw input of shape (N, C, H, W)

        Returns:
            output (ndarray): The averaged segmentation output
        """
        # Get the correct device
        device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

        # Initialize empty activations list
        xs = []

        for m, c in zip(self.ms, self.cs):
            # Get the activations of an individual model `m`
            a = m(x.clone().repeat(1, c, 1, 1))
            a = F.softmax(a, dim=1).detach().cpu().numpy()
            xs.append(a)

        # Concat all outputs and average over
        xs = np.array(xs)
        output = torch.tensor(np.mean(xs, axis=0)).to(device)

        return output
```

2.2 Performance Analysis Mechanisms

To measure the model performance on the given dataset we implemented dataset representation, defined multiple choices of optimizers, loss functions, implemented the training and evaluation loops. Further details on the final choices and the theory behind certain configurations is explained in the **Experiments** section.

Dataset

In [ ]:

```
class ACDCDataset(torch.utils.data.Dataset):
    """ACDC dataset representation.

    This class reads the ACDC images and stores them. It can also apply
    transformations while reading the images if the transform pipeline
    is provided.
    """

    def __init__(self, paths='', mask_exists=True, transform=None, n_aug=0):
        """Initializes the dataset file.
        Note:
            The path to the dataset directory is expected to contain two
            subdirectories: `image` and `mask` where original images and
            corresponding masks are stored. If the `mask` directory does
            not exist, there is an opportunity to store just the images.

        Args:
            paths (str | tuple(str)): The path(-s) to the dataset folder(-s)
            mask_exists (bool): Whether the masks are present
            transform (): Augmentation to apply
        """
        super().__init__()

        # Set the dataset parameters
        self.mask_exists = mask_exists
        self.transform = transform

        # Initialize image and mask lists
        self.img_files = []
        self.mask_files = []

        if isinstance(paths, str):
            # If it's a single path, convert to tuple
            paths = [paths]

        for path in paths:
            # Read the original image files and initialize mask list
            img_files = glob(os.path.join(path, "image", "*.png"))
            mask_files = []

            if mask_exists:
                # If masks exist, read all the mask files
                for img_path in img_files:
                    # Get the mask image file name
                    base = os.path.basename(img_path)
                    mask = os.path.join(path, 'mask', base[:-4] + '_mask.png')

                    # Append the mask file to the list
                    mask_files.append(mask)

            # Append image and mask files to the full list
            self.img_files += img_files
            self.mask_files += mask_files

        # Generate image and mask values
        self.images = [cv2.imread(f, cv2.IMREAD_UNCHANGED) for f in self.img_files]
        self.masks = [cv2.imread(f, cv2.IMREAD_UNCHANGED) for f in self.mask_files]

        # Initialize augmentation lists
        img_augs = []
        mask_augs = []

        for _ in range(n_aug):
            if mask_exists and self.transform is not None:
                for img, msk in zip(self.images, self.masks):
                    # Apply augmentation for images and masks
                    augmentation = self.transform(image=img, mask=msk)
                    img_augs.append(augmentation["image"])
                    mask_augs.append(augmentation["mask"])
            elif self.transform is not None:
                for img in self.images:
                    # Apply augmentation for images
                    augmentation = self.transform(image=img)
                    img_augs.append(augmentation["image"])

        # Append augmented images and masks to original ones
        self.images += img_augs
        self.masks += mask_augs


    def __getitem__(self, index):
        """Gets the image(s) at the requested index as float array.

        Args:
            index (int): The index of the desired image

        Returns:
            (tuple): A pair of image arrays, one representing the
                     original image, the other representing the mask.
                     If there are no masks, a single data array instead
                     of a tuple is returned.
        """
        # Get the image at the specified index
        image = self.images[index]

        if not self.mask_exists:
            # If mask doesn't exist, it's just a single image
            return torch.from_numpy(image).float()

        # Get the massk at the specified index
        mask = self.masks[index]

        return torch.from_numpy(image).float(), torch.from_numpy(mask).float()


    def __len__(self):
        """Gets the length of the dataset.
        Return:
            (int): The number of images the dataset contains
        """
        return len(self.images)


def get_transform(augment=True, normalize=True, full=False, composes=None):
    """Gets the sequence of transformations

    Args:
        augment (bool): Whether augmentation should be applied
        normalize (bool): Whether normalization should be applied
        full (bool): Whether the training is done with validation data
    Returns:
        (A.Compose): A sequence of augmentations to apply
    """
    # Initialize the sequence of compositions
    compose = []

    if augment:
        if composes is None:
            # Append augmentations to be applied
            compose += [
                A.HorizontalFlip(p=0.5),
                A.VerticalFlip(p=0.5),
                A.RandomRotate90(p=0.5),
                A.ShiftScaleRotate(shift_limit=0.15, rotate_limit=0, border_mode=cv2.BORDER_REPL
ICATE, p=0.2),
            ]
        else:
            # Append custom augmentations
            compose += composes

    if normalize:
        # Get the precalculated values for mean and std
        mean = 70.7673 if full else 68.1232
        std = 61.2449 if full else 61.0349

        # Append the augmentation for compose
        compose += [A.Normalize(mean=[mean], std=[std])]

    return A.Compose([])


def get_data_loader(
        dataset_path,
        batch_size=16,
        num_workers=0,
        shuffle=True,
        augment=True,
        normalize=True,
        n_aug=0,
        full=False,
        mask_exists=True,
        composes=None
    ):
    """Gets the data loader.

    Args:
        dataset_path (str): The path to the dataset with `image` dir
        batch_size (int): The batch size
        num_workers (int): The number of workers on the loader
        shuffle (bool): Whether to shuffle the loaded entires
        augment (bool): Whether augmentation should be applied
        normalize (bool): Whether normalization should be applied
        n_aug (int): The number of extra augmented datasets to append
        full (bool): Whether the training is done with validation data
        mask_exists (bool): Whether the `mask` subdirectory exists
        composes (list): The list of augmentation composes

    Returns:
        (DataLoader): The data loader for training/evaluation
    """
    # Get the augmentation to apply
    transform = get_transform(augment, normalize, full, composes)

    # Create the dataset and the data loader
    dataset = ACDCDataset(dataset_path, mask_exists, transform, n_aug)
    loader = DataLoader(dataset, batch_size, shuffle, num_workers=num_workers)

    return loader
```

Configuration

In [ ]:

```python
def get_config(model, optimizer, epochs=100, batch_size=32, loss="CrossEntropy"):
    """Gets a configuration dictionary for the specified parameters

    Args:
        model (dict): The dictionary containing model information (e.g., name)
        optimizer (dict): The dictionary containing optimizer info (e.g., name)
        epochs (int): The number of epochs
        batch_size (int): The batch size
        loss (str): The name of the loss function to apply
    Returns:
        (dict): A dictionary containing model parameters
    """
    # Describe the default configuration
    config = {
        "epochs": epochs,
        "batch_size": batch_size,
        "dataset_info": {
            "train_data_path": "data/train",
            "val_data_path": "data/val",
            "test_data_path": "data/test"
        },
        "model_info": model,
        "optimizer_info": optimizer,
        "criterion_info": {
            "name": loss,
        }
    }
    return config

def load_config(path):
    """Loads model configuration.

    Returns:
        (dict): A dictionary containing model parameters
    """
    # Extract data from json file
    with open(path) as json_file:
        config = json.load(json_file)

    return config


def create_ensamble(configs):
    """Creates the model ensamble of n different models.

    Args:
        infos (list): List of model configurations

    Returns:
        (nn.Module): A model ensemble (segmentation voter)
    """
    # Check the device
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    # Initialize models and channels
    ms = []
    cs = []

    for config in configs:
        # Load the trained model
        m = build_model(config["model_info"]).to(device)
        m.load_state_dict(torch.load(config["model_info"]["model_path"]))

        # Append the model and the channel to the list
        ms.append(m)
        cs.append(config["model_info"]["input_channels"])

    return VotingSegmenter(ms, cs)


def build_model(model_info):
    """Builds a model based on the provided parameters.

    Args:
        model_info (dict): The model parameters

    Returns:
         (nn.Module): A model built on the provided specifications
    """
    if model_info["name"] == "FCN":
        # Generate FCN model
        model = FCN8s(
            pretrained_net=VGGNet(requires_grad=True),
            n_class=4
        )
    elif model_info["name"] == "UNet":
        # Generate UNet model
        model = UNet(
            in_channels=model_info["input_channels"],
            depth=model_info["depths"],
            num_classes=4
        )
    elif model_info["name"] == "SegNet":
        # Generate SegNet model
        model = SegNet(
            input_channels=model_info["input_channels"],
            output_channels=4,
        )
    elif model_info["name"] == "DeepLab_v3":
        # Generate DeepLab model
        model = DeepLab(
            backbone="drn",
            output_stride=16,
            num_classes=4
        )
    elif model_info["name"] == "EffUNet":
        # Generate EffUNet model
        model = get_efficientunet_b3(
            out_channels=4,
            concat_input=True,
            pretrained=True
        )
    elif model_info["name"] == "Ensemble":
        # Generate model ensemble
        model = create_ensamble(model_info["configs"])
    else:
        raise ValueError(f"Model {model_info['name']} is invalid!")

    return model
```

```python
def build_optimizer(model, optimizer_info):
    """Builds an optimizer based on the provided parameters.

    Args:
        model (nn.Module): The model whose parameters to update
        model_info (dict): The optimizer parameters

    Returns:
         (optim) An optimizer built on the provided specifications
    """
    if optimizer_info["name"] == "Adam":
        # Generate Adam optimizer
        optimizer = optim.Adam(
            model.parameters(),
            lr=optimizer_info["lr"],
            weight_decay=optimizer_info["weight_decay"]
        )
    elif optimizer_info["name"] == "RMSprop":
        # Generate RMSProp optimizer
        optimizer = optim.RMSprop(
            model.parameters(),
            lr=optimizer_info["lr"],
            weight_decay=optimizer_info["weight_decay"],
            momentum=optimizer_info["momentum"]
        )
    elif optimizer_info["name"] == "SGD":
        # Generate SGD optimizer
        optimizer = optim.SGD(
            model.parameters(),
            lr=optimizer_info["lr"],
            momentum=optimizer_info["momentum"],
            weight_decay=optimizer_info["weight_decay"],
            nesterov=True
        )
    else:
        raise ValueError(f"Optimizer {optimizer_info['name']} is invalid!")

    return optimizer


def build_criterion(criterion_info):
    """Builds a criterion based on the provided parameters.

    Args:
        criterion_info (dict): The loss parameters

    Returns:
         (nn.Module): A criterion built on the provided specifications
    """
    if criterion_info["name"] == "CrossEntropy":
        # Generate Cross Entropy criterion
        criterion = nn.CrossEntropyLoss()
    elif criterion_info["name"] == "Dice":
        # Generate Dice criterion
        criterion = DiceLoss(to_onehot_y=True, softmax=True, jaccard=True)
    elif criterion_info["name"] == "DiceCE":
        # Generate Dice Cross Entropy criterion
        criterion = DiceCELoss(to_onehot_y=True, softmax=True, jaccard=True)
    elif criterion_info["name"] == "DiceFocal":
        # Generate Dice Focal criterion
        criterion = DiceFocalLoss(to_onehot_y=True, softmax=True)
    elif criterion_info["name"] == "Focal":
        # Generate Focal criterion
        criterion = FocalLoss(to_onehot_y=True)
    elif criterion_info["name"] == "Tversky":
        # Generate Tversky criterion
        criterion = TverskyLoss(to_onehot_y=True, softmax=True, include_background=False)
    else:
        raise ValueError(f"Criterion {criterion_info['name']} is invalid!")

    return criterion


def prepare_for_loss(loss_name, target):
    """Prepares the target for the loss function.

    Args:
        loss_name (str): The name of the loss funciton
        target (torch.tensor): The target values of shape (N, H, W)

    Returns:
        (torch.tensor): The prepared target tensor
    """
    # Loss functions that require channel dimension
    REQUIRES_CHANNEL_DIM = [
        "Dice",
        "DiceCE",
        "DiceFocal",
        "Focal",
        "Tversky",
        "FocalTversky",
        "Contrastive"
    ]

    if loss_name in REQUIRES_CHANNEL_DIM:
        # Append extra channel dimension
        target = target.unsqueeze_(1)

    return target
```

Utilities

In [ ]:

```
def get_config_args(config, device):
    """Gets the configuration arguments based on the config dictionary.

    Args:
        config (dict): The dictionary containig configuration files
        device (torch.device): The device the calculations are done on

    Returns:
        (tuple): A tuple of constructed arguments
    """
    # Get the model, optimizer and criterion from the config dictionary
    model = build_model(config["model_info"]).to(device)
    optimizer = build_optimizer(model, config["optimizer_info"])
    criterion = build_criterion(config["criterion_info"]).to(device)

    # Get the training parameters from the config dictionary
    epochs = config["epochs"]
    batch_size = config["batch_size"]

    # Get the dataset info from the config dictionary
    train_path = config['dataset_info']["train_data_path"]
    val_path = config['dataset_info']["val_data_path"]

    return model, optimizer, criterion, epochs, batch_size, train_path, val_path


def plot_train_val(train_losses_list, val_losses_list, labels):
    """Plots train vs validation losses in different subplots.

    Args:
        train_losses_list (list): The list of list of training losses
        val_losses_list (list): The list of list of validation losses
        labels (list): The list of labels for each param choice
    """
    # Convert to numpy array
    train_losses_list = np.array(train_losses_list)
    val_losses_list = np.array(val_losses_list)

    # Prepare for plotting
    x = np.arange(train_losses_list.shape[1])
    rows = (len(labels)+1) // 2
    plt.figure(figsize=(10, 15))

    for i in range(len(labels)):
        # Set the plot constraints
        plt.subplot(rows, 2, i+1)
        plt.ylim([0, 2])
        plt.xlim([0, len(x)])

        # Plot the graph
        plt.plot(x, train_losses_list[i, :], label ="Train Loss")
        plt.plot(x, val_losses_list[i, :], label ="Val Loss")

        # Set the subplot descriptions
        plt.xlabel("Epochs")
        plt.title(labels[i])
        plt.legend()

    # Show the plot
```

```python
    plt.show()


def plot_losses(train_losses_list, val_losses_list, labels, loss_name="Loss"):
    """Plots the loss comparisons.

    Args:
        train_losses_list (list): The list of list of training losses
        val_losses_list (list): The list of list of validation losses
        labels (list): The list of labels for each param choice
        loss_name (str): The  name of the loss
    """
    # Convert to numpy array
    train_losses_list = np.array(train_losses_list)
    val_losses_list = np.array(val_losses_list)

    # Prepare for plotting
    y_max = 1.5
    fig, axes = plt.subplots(1, 2, figsize=(15, 5))
    x = np.arange(train_losses_list.shape[1])

    for i, label in enumerate(labels):
      # Plot the values in subplots
      axes[0].plot(x, train_losses_list[i, :], label=label)
      axes[1].plot(x, val_losses_list[i, :], label=label)

    for i in range(len(axes)):
        # Set the subplot descriptions
        axes[i].set_ylim([0, y_max])
        axes[i].set_xlim([0, len(x)])
        axes[i].legend(loc="upper right")
        axes[i].set_xlabel("Epochs")
        axes[i].set_ylabel(loss_name)

    # Set the subplot titles
    axes[0].set_title("Train Loss")
    axes[1].set_title("Val Loss")

    # Plot the results
    plt.show()


def plot_comparisons(config_callback, param_choices, labels, loss_name="Loss", train_val=False):
    """Performs training on different parameters and plots the result.

    Args:
        config_callback (function): The function which produces a config file
        param_choices (list): The list of parameter choices the config depends on
        labels (list): The list of labels for each param choice
        loss_name (str): The  name of the loss
        train_val (bool): Whether the sublots have train and validation

    Returns:
        (tuple): A list of lists of models, train and eval losses
    """
    # Initialize list of list of models and losses
    models = []
    train_losses_list = []
    val_losses_list = []

    for i, param_choice in enumerate(param_choices):
        # Get config and train
        config = config_callback(param_choice)
        model, train_losses, val_losses = train(config)

        # Append model and list of losses
        models.append(model)
        train_losses_list.append(train_losses)
        val_losses_list.append(val_losses)

    if train_val:
        # Plot the loss comparisons within same plots
        plot_train_val(train_losses_list, val_losses_list, labels)
    else:
        # Plot the loss comparisons in separate plots
        plot_losses(train_losses_list, val_losses_list, labels, loss_name)

    return models, train_losses_list, val_losses_list


def save_model(model, config, epoch=None):
    """Saves the trained model to the provided dir.

    Args:
        model (nn.Module): The trained model
        config (dict): The parameter dictionary with model path
        epoch (int): The nth epoch the model was trained
    """
    # Get path to the model
    path = config['model_info']['model_path']

    if epoch is not None:
        # If epoch is provided, change the name
        path = path[:-4] + f"_ep{epoch}.pth"

    if not os.path.exists("models"):
        # Create `models` dir if it not exists
        os.makedirs("models")

    # Save the model
    torch.save(model.state_dict(), path)


def save_losses(train_losses, val_losses, config):
    """Saves the monitored losses to `models` dir.

    Args:
        train_losses (list(float)): The training losses
        val_losses (list(float)): The validation losses
        config (dict): The parameter dictionary with model path
    """
    # Get model path and optimizer name
    model_path = config["model_info"]["model_path"][:-4]
    op_name = config["optimizer_info"]["name"]

    with open(f"{model_path}_{op_name}_loss.npz","wb") as f:
        # Save the loss history to model path
        d = {'file': f, 'train_losses': train_losses, 'val_losses': val_losses}
        np.savez(**d)


def get_dice_score(output, y):
    """Calculates the dice score.

    Args:
        output (torch.tensor): The raw network outputs of shape (N, C, H, W)
        y (torch.tensor): The true mask of shape (N, H, W)

    Returns:
        (torch.tensor): The (1 - loss) values of shape (C - 1,)
    """
    # Get the raw predictions
    _, y_hat = torch.max(output, 1)
    y_hat = one_hot(y_hat.unsqueeze_(1), num_classes=output.shape[1])

    # Define the dice loss (ignore first channel)
    dice_loss_fn = DiceLoss(to_onehot_y=True, include_background=False)

    return 1 - dice_loss_fn(y_hat, y)


def rle_encoding(x):
    """Encodes an image.

    Note:
        Image is represented by 1s (mask) and 0s (background).
        Credit to https://www.kaggle.com/rakhlin/fast-run-length-encoding-python

    Args:
        x (ndarray): The image representation of shape (H, W)

    Returns:
        (list): A list of run lengths
    """
    dots = np.where(x.T.flatten() == 1)[0]
    run_lengths = []
    prev = -2
    for b in dots:
        if (b > prev + 1): run_lengths.extend((b + 1, 0))
        run_lengths[-1] += 1
        prev = b
    return run_lengths


def submission_converter(mask_directory, path_to_save):
    """Converts mask images to a submission file.

    Args:
        mask_directory (str): The path to the `mask` image directory
        path_to_save (str): The path to where the submission file is saved
    """
    writer = open(path_to_save, 'w')
    writer.write("id,encoding¥n")

    files = os.listdir(mask_directory)

    for file in files:
        name = file[:-4]
        mask = cv2.imread(os.path.join(mask_directory, file), cv2.IMREAD_UNCHANGED)

        mask1 = (mask == 1)
        mask2 = (mask == 2)
        mask3 = (mask == 3)
        encoded_mask1 = rle_encoding(mask1)
        encoded_mask1 = ' '.join(str(e) for e in encoded_mask1)
        encoded_mask2 = rle_encoding(mask2)
        encoded_mask2 = ' '.join(str(e) for e in encoded_mask2)
        encoded_mask3 = rle_encoding(mask3)
        encoded_mask3 = ' '.join(str(e) for e in encoded_mask3)

        writer.write(name + '1,' + encoded_mask1 + "¥n")
        writer.write(name + '2,' + encoded_mask2 + "¥n")
        writer.write(name + '3,' + encoded_mask3 + "¥n")

    writer.close()
```

Training

In [ ]:

```
def prepare_values(x, y, device, config):
    """Prepares the values for training/evaluation

    Args:
        x (torch.tensor): The input tensor
        y (torch.tensor): The target tensor
        device (torch.device): The device to run the step function on
        config (dict): The parameter dictionary
    """
    # Load the tensors to the proper device
    x, y = x.to(device), y.to(device, dtype=torch.long)

    # Ensure channel dimension of size `1` exists
    x.unsqueeze_(1)

    if config["model_info"]["input_channels"] > 1:
        # Expand channels if the model works on more than 1 channel
        x = x.repeat(1, config["model_info"]["input_channels"], 1, 1)

    # Loss function may expect or not expect the empty channel dimension
    y = prepare_for_loss(config["criterion_info"]["name"], y)

    return x, y


def step(data_loader, model, optimizer, criterion, device, config, eval=False):
    """Performs a step function for training/evaluating the model

    Args:
        data_loader (DataLoader): The dataset loader with x and y values
        model (nn.Module): The model for the data
        optimizer (optim.Optimizer): The optimizer
        criterion (nn.Module): The loss function
        device (torch.device): The device to run the step function on
        config (dict): The parameter dictionary
        eval (bool): Whether the model is being trained or evaluated

    Returns:
         (float): The total loss
    """
    if eval:
        # If `eval`, set the model to eval mode
        model.eval()
    else:
        # Otherwise set the model to train mode
        model.train()

    # Initialize the total loss
    total_loss = 0

    for x, y in data_loader:
        # Reset the gradient
        optimizer.zero_grad()

        # Ensure valid values
        x, y = prepare_values(x, y, device, config)

        # Get the final activations
        output = model(x)
        # Calculate the loss
        loss = criterion(output, y)

        if not eval:
            # Calculate the parameter gradients
            loss.backward()

            # Perform the parameter update
            optimizer.step()

        # Add the loss to the total loss
        total_loss += loss.item()

    return total_loss


def train(config,
          n_aug=1,
          augment=False,
          composes=None,
          use_val=False,
          save_loss=False,
          save_model=False,
          save_checkpoints=False
    ):
    """Performs the traing for the provided model configuration.

    Args:
        config (dict): The dictionary containig configuration files
        n_aug (int): The number of augmentations to apply
        augment (bool): Whether to apply augmentation
        composes (list): The list of augmentation composes
        use_val (bool): Whether the validation set is included in training
        save_loss (bool): Whether the loss history should be saved to file
        save_model (bool): Whether the trained model should be saved
        save_checkpoints (bool): Whether model checkpoints should be saved

    Returns:
        tuple: A tuple which includes the model, and list of train and val losses
    """
    # Get the correct device
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    # Use 0 workers for Windows
    if os.name == "nt":
        num_workers = 0
    else:
        num_workers = 2

    # Get the configutration arguments
    model, optimizer, criterion, epochs, batch_size, train_path, val_path = get_config_args(conf
ig, device)

    # Define common parameters for compact notation
    common = batch_size, num_workers

    if use_val:
        # If full training is performed, use train and validation sets
        full_path = train_path, val_path
        train_loader = get_data_loader(full_path, batch_size, num_workers, full=True, augment=au
gment, n_aug=n_aug, composes=composes)
    else:
        # Load training data and validation data
        train_loader = get_data_loader(train_path, batch_size, num_workers, augment=augment, n_a
ug=n_aug, composes=composes)
        val_loader = get_data_loader(val_path, batch_size, num_workers, augment=False)

    # Initialize the list of train and validation losses
    train_losses, val_losses = [], []

    # Initialize loss values (for the tbar)
    train_loss = -1
    val_loss = -1

    # Initialize the loading bar
    tbar = tqdm(range(epochs))
    tbar.set_description(f"[0] Train: N/A | Val: N/A")

    for epoch in tbar:
        # Calculate the training loss and append it to history
        train_loss = step(train_loader, model, optimizer, criterion, device, config)
        train_losses.append(train_loss)

        if not use_val:
            # If validation set is not used for training, record loss
            val_loss = step(val_loader, model, optimizer, criterion, device, config, eval=True)
            val_losses.append(val_loss)

        if save_checkpoints and (epoch + 1) % 10 == 0:
            # If checkpoints are required, save model's checkpoints
            save_model(model, config, epoch + 1)

        # Update the description of the progress bar
        tbar.set_description(f"[{epoch+1}] Train: {train_loss:.4f} | Val: {val_loss:.4f}")

    if save_model:
        # Save the trained model
        save_model(model, config)

    if save_loss:
        # Save loss history if needed
        save_losses(train_losses, val_losses, config)

    return model, train_losses, val_losses
```

Evaluation and Prediction

In [ ]:

```
def evaluate(val_loader, model, device, config):
    """Evaluates the model on the validation set.

    Args:
        val_loader (DatasetLoader): The loader for the validation set
        model (nn.Module): The model for the data
        device (torch.device): The device to run the step function on
        config (dict): The parameter dictionary
    """
    # Initialize the average loss
    mean_loss = 0

    for x, y in val_loader:
        # Ensure valid values
        config["criterion_info"]["name"] = "Dice"
        x, y = prepare_values(x, y, device, config)

        # Get the final activations
        output = model(x)
        _, pred = torch.max(output, 1)

        # Append dice accuracy to the overall accuracy
        mean_loss += get_dice_score(output, y)

    # Calculate the actual average loss
    mean_loss /= len(val_loader.dataset)

    # Print the average loss
    print(f"Average dice loss on the validation set = {mean_loss}")


def predict(test_loader, model, device, config):
    """Generates the predictions for the test set

    Args:
        val_loader (DatasetLoader): The loader for the validation set
        model (nn.Module): The model for the data
        device (torch.device): The device to run the step function on
        config (dict): The parameter dictionary
    """
    for i, x in enumerate(test_loader):
        # Get the valid input parameters
        x = x.to(device)
        x.unsqueeze_(1)
        x = x.repeat(1, config["model_info"]["input_channels"], 1, 1)

        # Generate the prediction
        output = model(x)
        _, pred = torch.max(output, 1)

        # Save the predicted masks to `test/mask` dir
        test_mask_path = config["dataset_info"]["test_data_path"] + "/mask"
        pred_mask_path = os.path.join(test_mask_path, f"cmr{121 + i}_mask.png")
        cv2.imwrite(pred_mask_path, pred.cpu()[0, ...].numpy())


def eval(config, use_val=False, ensemble=False):
    """Runs evaluation for the provided model configuration.
    Args:
        config (dict): The dictionary containig configuration files
        use_val (bool): Whether validation set was used for training
        ensemble (bool): Whether the configuration is for an ensemble model
    """
    # Get the correct device
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    # Use 0 workers for Windows
    if os.name == "nt":
        num_workers = 0
    else:
        num_workers = 2

    # Get the configutration arguments
    model, _, _, _, _, _, val_path = get_config_args(config, device)

    if not ensemble:
        # Load weights to the model if it's not ensamble
        model_path = config["model_info"]["model_path"]
        model.load_state_dict(torch.load(model_path))

    # Set model's state to `eval`
    model.eval()

    if use_val:
        # Load test data with batch size `1` and generate results
        test_path = config["dataset_info"]["test_data_path"]
        test_loader = get_data_loader(test_path, 1, num_workers, shuffle=False, mask_exists=Fals
e, augment=False, full=True)
        predict(test_loader, model, device, config)

        # Save the predictions to submission file
        pred_path = os.path.join(test_path, "mask")
        subm_path = config["model_info"]["name"] + "_submission.csv"
        submission_converter(pred_path, subm_path)
        print("Submission file saved.")
    else:
        # Load val data with batch size `1` and print results
        val_loader = get_data_loader(val_path, 1, num_workers, augment=False, shuffle=False)
        evaluate(val_loader, model, device, config)
```

3 Experiments

To improve the performance of the model, experiments are carried out to optimise the model for the semantic segmentation task given. This section describes the methods used to achieve the aforementioned purpose. This section will include the following:

Description on how the training data set, validation data set and test data set are used when training our model. Briefly describe each of the models that are used. Discussion of the justification to reasons for choosing/ not choosing the above models as the final prediction model used. Description of hyperparameter tuning process

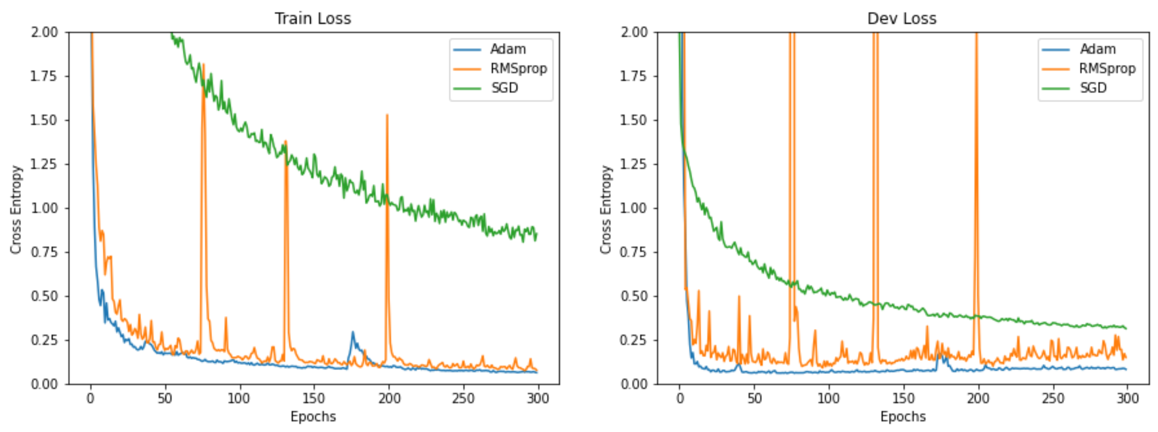
3.1 Optimizer and Loss

Optimizer Type

This experiment below is also done while considering DeepLabV3+ with pre-trained DRN as backbone. This model is trained with 3 different optimisers, namely Adaptive Moment Estimation (Adam), Root Mean Squared Propagation (RMSprop), and Stochastic Gradient Descent (SGD). After comparing different optimisers in parallel by plotting all of their performance in the same graph, it can be observed that the Adam optimisation algorithm with a learning rate of 0.001, Batch size 32, cross entropy as loss function has the best performance among all of the optimisation algorithm tested which the chosen architecture for this specific task, while keeping all other parameters (including hyperparameters) unchanged. Adam achieved the lowest training loss (left) (SGD > RMSProp > Adam) and has the lowest and a stable validation loss (right) (SGD > RMSProp ~ Adam), so it is chosen as the optimiser used for the model.

In [ ]:

```
# Num epochs
EPOCHS = 300

def get_config_optim(optimizer):
    """Gets the configuration dictionary.

    Args:
        optimizer (dict): The dictionary describing the optimizer

    Returns:
        (dict): Full config dictionary on which the training depends
    """
    # Describe the model configuration
    model = {
        "name": "DeepLab_v3",
        "input_channels": 3,
        "model_path": "models/effunet_optim.pth"
    }

    return get_config(model, optimizer, epochs=EPOCHS, batch_size=32, loss="CrossEntropy")

# Define possible optimizer choices
OPTIMIZER_CHOICES = [
    {"name": "Adam", "lr": 1e-3, "weight_decay": 1e-7},
    {"name": "RMSprop", "lr": 1e-3, "weight_decay": 1e-7, "momentum": 0},
    {"name": "SGD", "lr": 1e-3, "weight_decay": 1e-7, "momentum": 0.9}
]

# Define the labels
LABELS = [choice["name"] for choice in OPTIMIZER_CHOICES]

# Plot the results
_ = plot_comparisons(get_config_optim, OPTIMIZER_CHOICES, LABELS, loss_name="CrossEntropy")
```

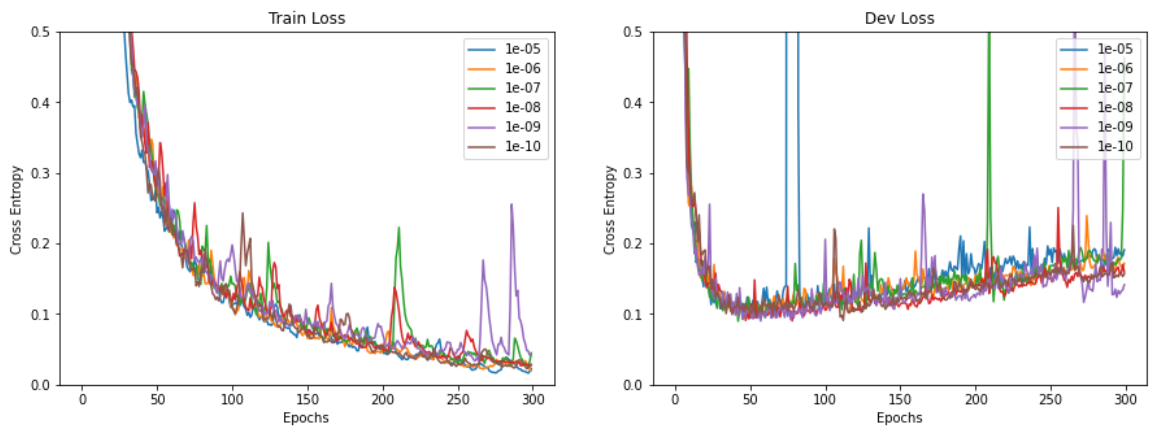


This experiment above is also done while considering DeepLabV3+ with pre-trained DRN as backbone. This model is trained with 3 different optimisers, namely Adaptive Moment Estimation (Adam), Root Mean Squared Propagation (RMSprop), and Stochastic Gradient Descent (SGD). After comparing different optimisers in parallel by plotting all of their performance in the same graph, it can be observed that the Adam optimisation algorithm with a learning rate of 0.001, Batch size 32, cross entropy as loss function has the best performance among all of the optimisation algorithm tested which the chosen architecture for this specific task, while keeping all other parameters (including hyperparameters) unchanged. Adam achieved the lowest training loss (left) (SGD > RMSProp > Adam) and has the lowest and a stable validation loss (right) (SGD > RMSProp ~ Adam), so it is chosen as the optimiser used for the model.

Regularization

In [ ]:

```
# Num epochs
EPOCHS = 300

def get_config_reg(reg):
    """Gets the configuration dictionary.

    Args:
        reg (float): The weight decay

    Returns:
         (dict): Full config dictionary on which the training depends
    """
    # Describe the model
    model = {
        "name": "EffUNet",
        "input_channels": 3,
        "model_path": "models/effunet_reg.pth"
    }

    # Describe the optimizer
    optimizer = {
        "name": "Adam",
        "lr": 0.001,
        "weight_decay": reg
    }

    return get_config(model, optimizer, epochs=EPOCHS, batch_size=32, loss="CrossEntropy")

# Define possible regularization choices
REG_CHOICES = [1e-5, 1e-6, 1e-7, 1e-8, 1e-9, 1e-10]

# Define the labels
REG_LABELS = [f"{reg:.0e}" for reg in REG_CHOICES]

# Plot the results
_ = plot_comparisons(get_config_reg, REG_CHOICES, REG_LABELS, loss_name="CrossEntropy")
```



As shown from the above figure, the choice of the regularisation term does not show great difference in performance apart from some instability after the point where the model is clearly overfitting. So the regularisation term is kept unchanged (1e-07).

Loss Type

The 2 loss functions that are used for the evaluation of the model are introduced briefly here.

---

1. Cross-Entropy Loss Function

The cross-entropy loss function is the main loss function used for evaluation of the model during training, as it has the most stable performance and produces good results.

The cross-entropy loss function measures how similar/ dissimilar 2 probability distributions are, which is defined as:

$$E(\mathbf{w}) = -\sum_{i=1}^{N} y^{(i)} \ln p_1(\mathbf{x}^{(i)}, \mathbf{w}) + (1 - y^{(i)}) \ln (1 - p_1(\mathbf{x}^{(i)}, \mathbf{w}))$$

where $y^{(i)}$ is the actual output of the i^{th} sample, $\mathbf{x}^{(i)}$ is the input of the i^{th} sample, and $\mathbf{w}$ is the current learned weights. Further details can be found here (https://en.wikipedia.org/wiki/Cross_entropy).

---

2. Dice Loss Function

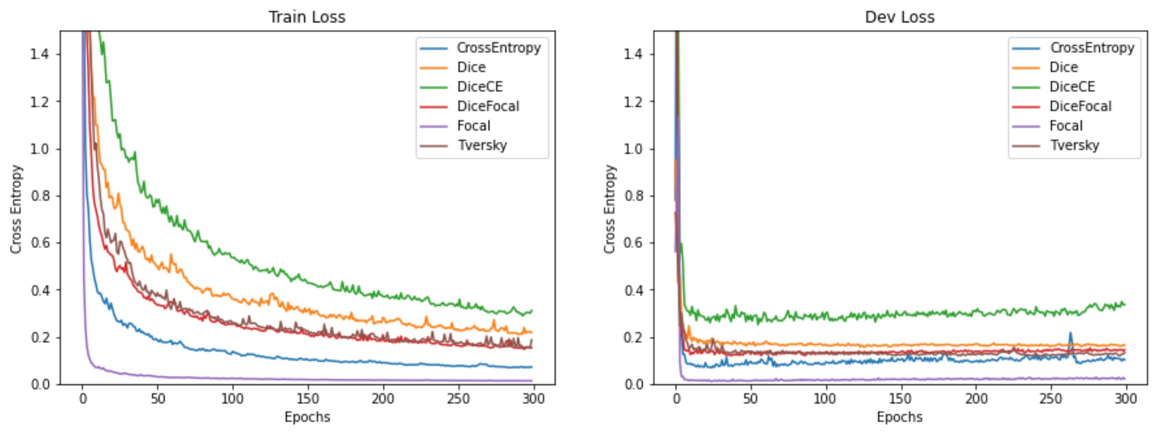
The dice loss function is the function used for evaluation on the Kaggle platform. However, as its performance is not as good as the cross entropy function as shown from the below figure it was not chosen for the model evaluation during training. The dice loss function is defined as below:

$$Dice(X, Y) = \frac{2|X \cap Y|}{|X| + |Y|}$$

Details of this loss function can be found in the coursework task description and here (https://en.wikipedia.org/wiki/S%C3%B8rensen%E2%80%93Dice_coefficient).

In [ ]:

```
# Num epochs
EPOCHS = 300

def get_config_loss(loss):
    """Gets the configuration dictionary.

    Args:
        loss (str): The name of the loss function

    Returns:
        (dict): Full config dictionary on which the training depends
    """
    # Describe the model
    model = {
        "name": "DeepLab_v3",
        "backbone": "drn",
        "input_channels": 3,
        "model_path": "models/deeplab_loss.pth"
    }

    # Describe the optimizer
    optimizer = {
        "name": "Adam",
        "lr": 0.001,
        "weight_decay": 1e-8
    }

    return get_config(model, optimizer, epochs=EPOCHS, batch_size=32, loss=loss)

# Define possible loss choices
LOSS_CHOICES = ["CrossEntropy", "Dice", "DiceCE", "DiceFocal", "Focal", "Tversky"]

# Define the labels
LOSS_LABELS = LOSS_CHOICES

# Plot the results
_ = plot_comparisons(get_config_loss, LOSS_CHOICES, LOSS_LABELS, loss_name="Various Loss")
```



As shown from the above graph, focal loss is able to reduce the loss the most. However, as the loss is very quickly reduced to near 0, there is a high risk of not recognising overfitting from the usage of focal loss function. After careful consideration and discussion, the loss function with second best performance, the cross entropy loss function is finally chosen, as it performs well in general and has a learning curve that is recognisable.

3.2 Data

The task has provided 3 data sets, a training set (100 samples), a validation set (20 samples) and a test set (80 samples). Both training set and validation set contains the original images and correctly segmented images, while the test set only contains the original, non-segmented images.

The weights are only updated and learned using the training set, where the validation set is for evaluation if the current learnt weights are overfitting to the training data, indicated by a decreasing training loss and an increasing validation loss. The test set is for making predictions to submit to the Kaggle platform for final evaluation of the model.

Augmentation

Since only 100 samples are given for training, some augmentation techniques are tried to enhance the sample. Some of the following techniques are carried out as experimentation:

- Horizontal Flip
- Vertical Flip
- 90 degree Rotation
- Adding noise
- Shifting
- Shear

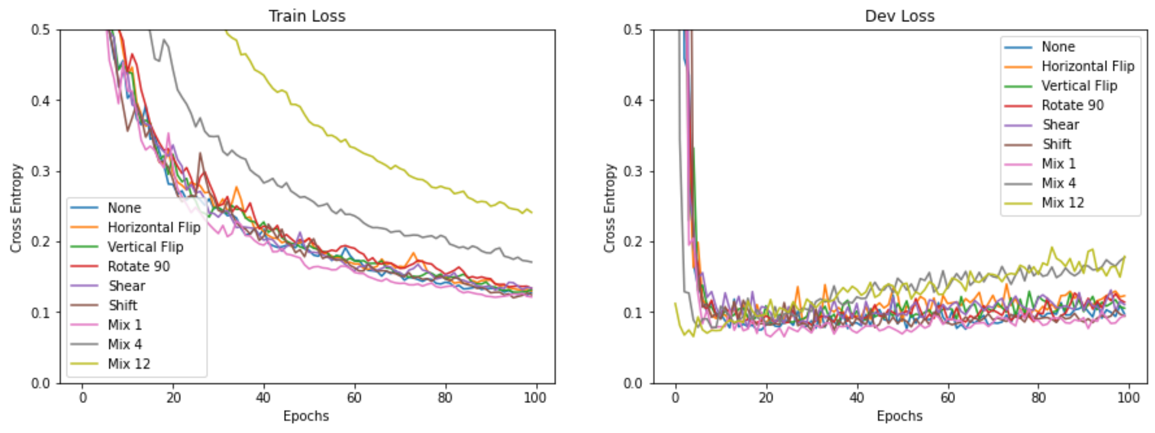
Type(EffUnet, epoch = 60)	Val_Accuracy (mean of 3)
None	0.871
Horizontal	0.864
Vertical	0.888
Rotate	0.872
Shift	0.865
Noise	0.835
Blur	0.841
Shear	0.873

To identify which techniques will help for improving performance, the above techniques are applied one by one using random probability and taking a mean. The test could potentially be biased since there is only limited time and machine power. As shown from the spreadsheet above, some features improved the accuracy of the test, and some did not. It can be observed that Vertical flip brings a higher accuracy but not for the Horizontal flip. Shifting will make the accuracy lower than without the augmentation. This could be caused by when the image has shifted, they will be broader with an unknown plane that it has to be covered with some generated pixels. The technique that gives the best accuracy, replication method, which copies the most edge pixel, and rotation method are chosen. Blur and adding noises came out worse than the original. The techniques that yield an accuracy below 0.86 were not chosen and parameters were tuned for the actual test based on the accuracy that this result gave.

In [ ]:

```
# Num epochs
EPOCHS = 100

# Compose choices
COMPOSES = [
    (1, []),
    (1, [A.HorizontalFlip(p=1)]),
    (1, [A.VerticalFlip(p=1)]),
    (1, [A.RandomRotate90(p=1)]),
    (1, [A.Affine(shear=0.3, p=1)]),
    (1, [A.ShiftScaleRotate(shift_limit=0.1, rotate_limit=0, border_mode=cv2.BORDER_REPLICATE, p
=.2)]),
    (1, [A.HorizontalFlip(p=0.3),
         A.VerticalFlip(p=0.6),
         A.RandomRotate90(p=0.2),
         A.OneOf([
            A.Affine(shear=0.3, p=0.8),
            A.ShiftScaleRotate(shift_limit=0.1, rotate_limit=0, border_mode=cv2.BORDER_REPLICATE
, p=.2)
         ], p=.3)]),
    (4, [A.HorizontalFlip(p=0.3),
         A.VerticalFlip(p=0.6),
         A.RandomRotate90(p=0.2),
         A.OneOf([
            A.Affine(shear=0.3, p=0.8),
            A.ShiftScaleRotate(shift_limit=0.1, rotate_limit=0, border_mode=cv2.BORDER_REPLICATE
, p=.2)
          ], p=.3)]),
    (12, [A.HorizontalFlip(p=0.3),
          A.VerticalFlip(p=0.6),
          A.RandomRotate90(p=0.2),
          A.OneOf([
            A.Affine(shear=0.3, p=0.8),
            A.ShiftScaleRotate(shift_limit=0.1, rotate_limit=0, border_mode=cv2.BORDER_REPLICATE
, p=.2)
          ], p=.3)])
]

# Define labels
LABELS = ["None", "Horizontal Flip", "Vertical Flip", "Rotate 90", "Shear", "Shift", "Mix 1", "M
ix 4", "Mix 12"]

# Describe the model
model = {
    "name": "DeepLab_v3",
    "backbone": "drn",
    "input_channels": 3,
    "model_path": "models/deeplab_loss.pth"
}

# Describe the optimizer
optimizer = {
    "name": "Adam",
    "lr": 0.001,
    "weight_decay": 1e-8
}

# Get the configuration file
config = get_config(model, optimizer, epochs=EPOCHS, batch_size=32)
```

```
# Initialize the empty lists of losses
train_losses_list = []
val_losses_list = []

for n_aug, composes in COMPOSES:
    # Generate train losses and validation losses after one training
    m, train_losses, val_losses = train(config, n_aug=n_aug, augment=True, composes=composes)

    # Append the losses to the lists
    train_losses_list.append(train_losses)
    val_losses_list.append(val_losses)

# Plot the results
plot_losses(train_losses_list, val_losses_list, LABELS, loss_name="CrossEntropy")
```



3.3 Process of Hyperparameter Tuning

For all the below experiments carried out for tuning hyperparameters, only the relevant hyperparameter is changed for each of the experiments, while all other parameters, including hyperparameters are kept the same as controlled variables during the experiment.

For optimisation purpose so the model can achieve a better performance, the following hyperparameters are tuned:

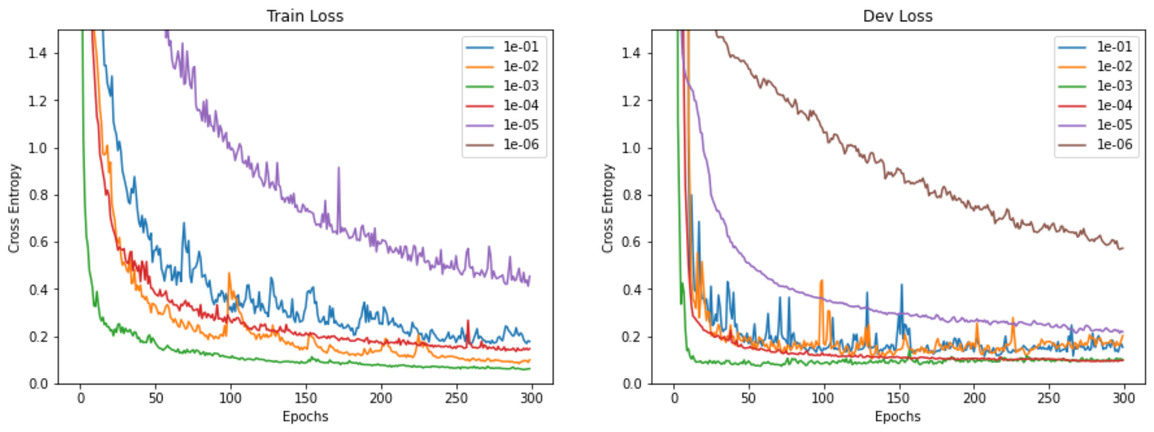
1. Learning rate - step size, where its value controls the strength of the loss, i.e. how much the loss is weighted in each update of the weights
2. Batch size - the number of data samples used in each training iteration
3. Epochs - the number of times that the model is trained on the entire training data set

The codes for the experiments are attached and we demonstrate the experiments processes by running it.

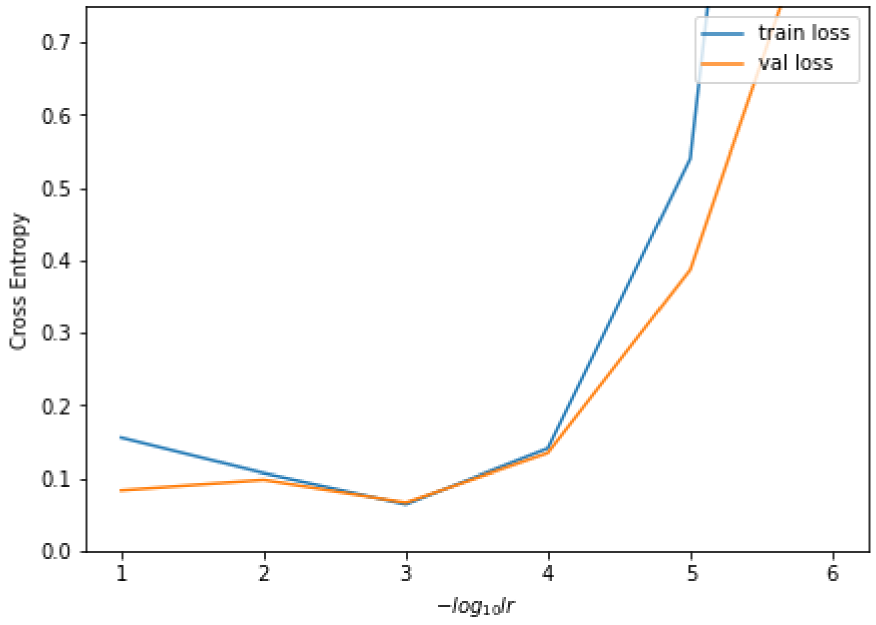
Learning Rate

In [ ]:

```
# Num Epochs
EPOCHS = 10

def get_config_lr(lr):
    """Gets the configuration dictionary.

    Args:
        lr (float): The learning rate

    Returns:
        (dict): Full config dictionary on which the training depends
    """
    # Describe the model
    model = {
        "name": "DeepLab_v3",
        "backbone": "drn",
        "input_channels": 3,
        "model_path": "models/deeplab_lr.pth"
    }

    # Describe the optimizer
    optimizer = {
        "name": "Adam",
        "lr": lr,
        "weight_decay": 1e-8
    }

    return get_config(model, optimizer, epochs=EPOCHS, batch_size=32, loss="CrossEntropy")

# Define possible learning rate choices
LR_CHOICES = [1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6]

# Define the labels
LR_LABELS = [f"{lr:.0e}" for lr in LR_CHOICES]

# Plot the results
_ = plot_comparisons(get_config_lr, LR_CHOICES, LR_LABELS, loss_name="CrossEntropy")
```



The above figure shows the effect of using different learning rates on the model (DeepLabV3+ with DRN backbone). It can be seen that the value of 1e-3, i.e. 10^{-3}/ 0.0001 has the best performance for both training and validation process.

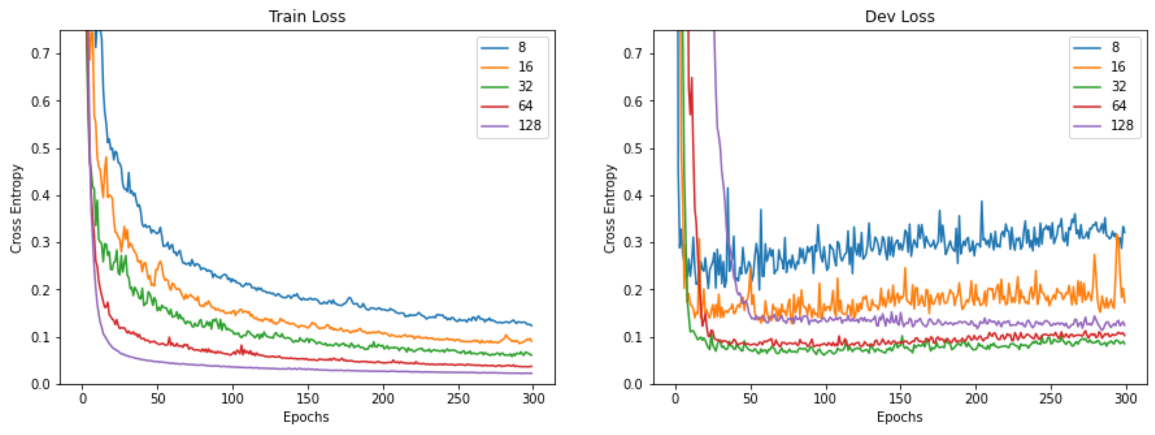


The above figure is a simplified version of the previous figure, which shows the lowest achieved training and validation loss using different values as the learning rate, where the lowest is achieved at -$\log_{10}$(learning rate) = 3, i.e. 10^{-3} / 0.0001 for both training loss and validation loss. Combining the findings from these 2 plots, the value of 10^{-3} / 0.0001 is chosen as the learning rate used for the model architecture.

Batch Size of Sample Data

In [ ]:

```
# Num epochs
EPOCHS = 300

def get_config_batch(batch_size):
    """Gets the configuration dictionary.

    Args:
        batch_size (int): The batch size

    Returns:
        (dict): Full config dictionary on which the training depends
    """
    # Describe the model
    model = {
        "name": "DeepLab_v3",
        "backbone": "drn"
        "input_channels": 3,
        "model_path": "models/deeplab_loss.pth"
    }

    # Describe the optimizer
    optimizer = {
        "name": "Adam",
        "lr": 0.001,
        "weight_decay": 1e-8
    }

    return get_config(model, optimizer, epochs=EPOCHS, batch_size=batch_size, loss="CrossEntrop
y")

# Define possible loss choices
BATCH_CHOICES = [8, 16, 32, 64, 128]

# Define the labels
BATCH_LABELS = list(map(str, BATCH_CHOICES))

# Plot the results
_ = plot_comparisons(get_config_batch, BATCH_CHOICES, BATCH_LABELS, loss_name="CrossEntropy")
```

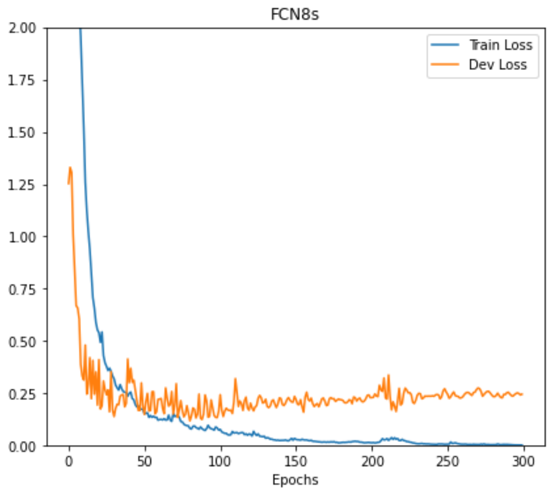


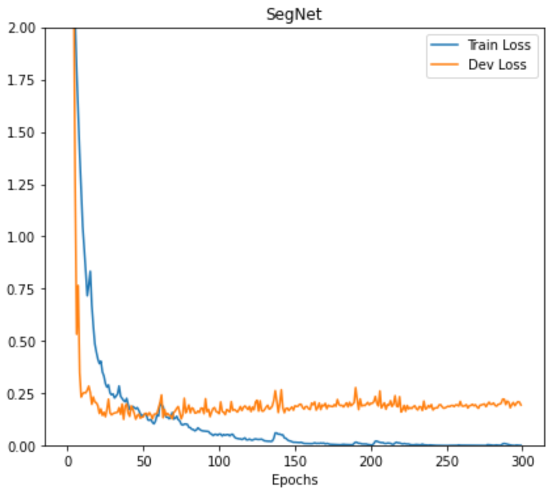
The above figure shows the comparison of loss for using different batch sizes of sample data when training the selected model, DeepLabV3+ with ResNet backbone. On the left shows the training loss and on the right shows the validation loss. The cross entropy loss is tested on batch sizes of 8, 16 ,32 ,64 and 128. It can be observed that the larger the batch size, the smoother the curve can be obtained for training loss, and the lower the training error is. It can also be seen that only the batch size of 32 (green line) shows promising results for both training and validation process (it has lowest validation loss), hence this batch size of 32 is selected for training the model.

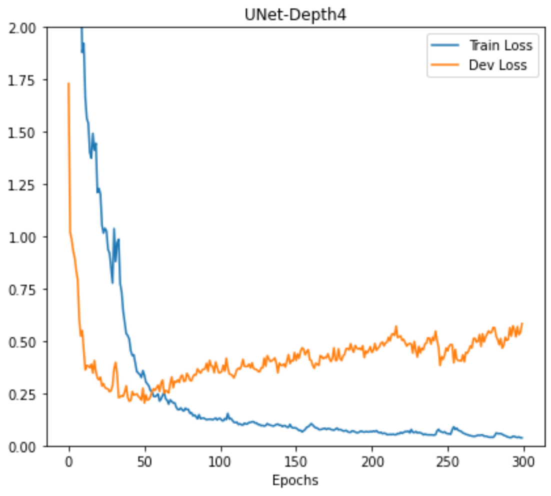
Number of Epochs

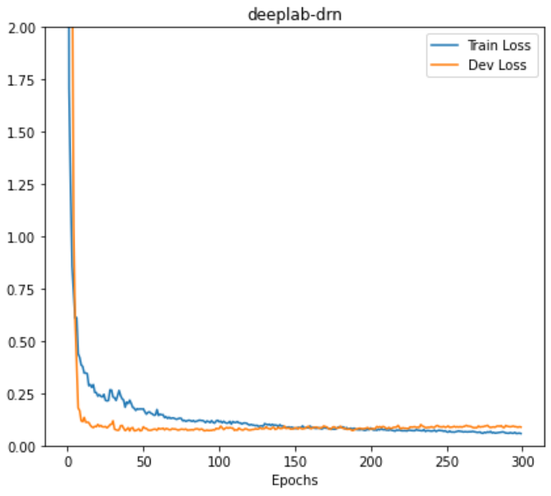
In [ ]:

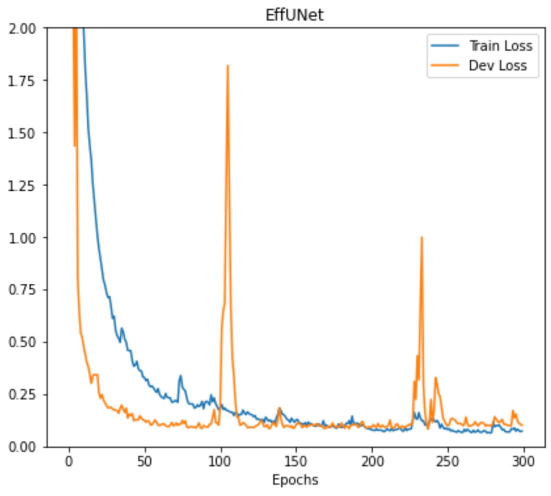
```
# Num epochs
EPOCHS = 300

def get_config_model(model):
    """Gets the configuration dictionary.

    Args:
        model (dict): The model description

    Returns:
        (dict): Full config dictionary on which the training depends
    """
    # Describe the optimizer
    optimizer = {
        "name": "Adam",
        "lr": 0.001,
        "weight_decay": 1e-7
    }

    return get_config(model, optimizer, epochs=EPOCHS, batch_size=32, loss="CrossEntropy")

# Define possible model choices
MODEL_CHOICES = [
    {"name": "FCN", "input_channels": 3, "model_path": "models/fcn_model.pth"},
    {"name": "SegNet", "input_channels": 3, "model_path": "models/segnet_model.pth"},
    {"name": "UNet", "depths": 4, "input_channels": 1, "model_path": "models/unet_model.pth"},
    {"name": "DeepLab_v3", "backbone": "drn", "input_channels": 3, "model_path": "models/deeplab
_model.pth"},
    {"name": "EffUNet", "input_channels": 3, "model_path": "models/effunet_model.pth"}
]

# Define the labels
MODEL_LABELS = [model["name"] for model in MODEL_CHOICES]

# PLot the results
_ = plot_comparisons(get_config_model, MODEL_CHOICES, MODEL_LABELS, loss_name="CrossEntropy", tr
ain_val=True)
```


FCN8s
Train Loss
Dev Loss
Epochs


SegNet
Train Loss
Dev Loss
Epochs


UNet-Depth4
Train Loss
Dev Loss
Epochs

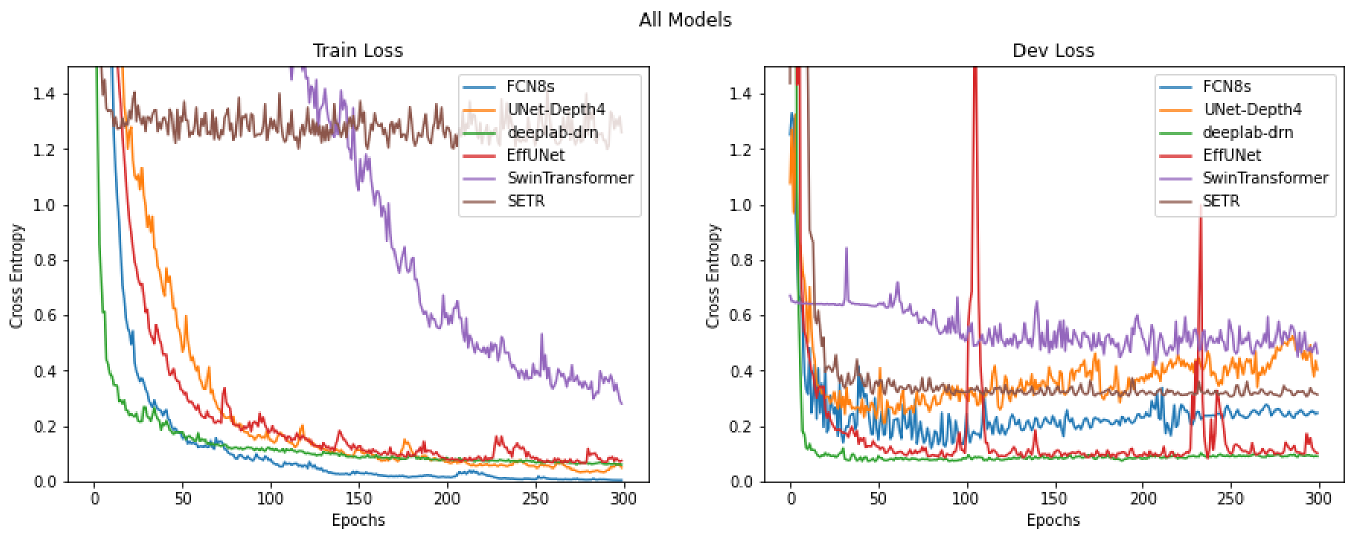

deeplab-drn
Train Loss
Dev Loss
Epochs


EffUNet
Train Loss
Dev Loss
Epochs

The above figure shows the visualisation of the effect of 0 - 300 epochs on the models that are experimented with. The validation error is usually lower than the training error in the early stage. However, it will surpass training error as the epoch increases, as the model is starting to overfit to the training data at a large epoch.

Eventually a point where training error meets the validation errors is chosen as the number of epochs to run for each model. It’s around 50 for FCN8s, 50 for SegNet, 55 for UNet-Depth4 and 150 for Deeplab-DRN, and 180 for EffUNet. It can be observed that the optimal number of epochs is dependent on the architecture, i.e. different architectures have different optimal number of epochs.

Performance Summarisation



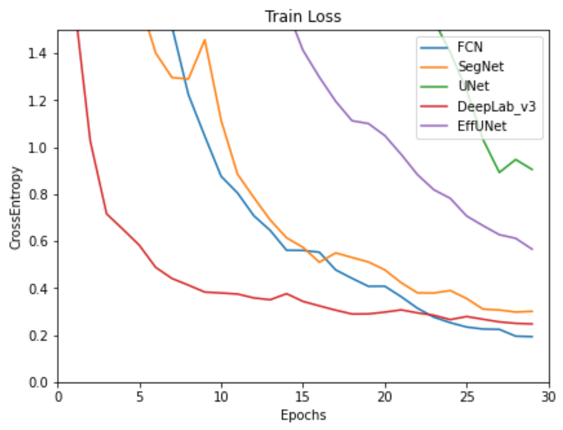
As shown from the above graph, models from UNet family(UNet, EffUnet), FCN family(FCNs, FCN8s, FCN32s, FCN8 is used here as it has the best performance among its family tested using similar method), DeeplabV3+ family (resnet, drn, mobilenet, xception backbones, drn is used here, see section 3.2.4 for choice justification details), and Transformer family(Swin, SETR) are all tested on the given task. Then typical models from the above classes are selected and optimised, their Cross Entropy Loss is plotted in the above figure. It shows how using different model types affects the loss. The training loss is shown on the left and the validation loss is shown on the right, and 6 models are plotted in different colors. It can be seen that:

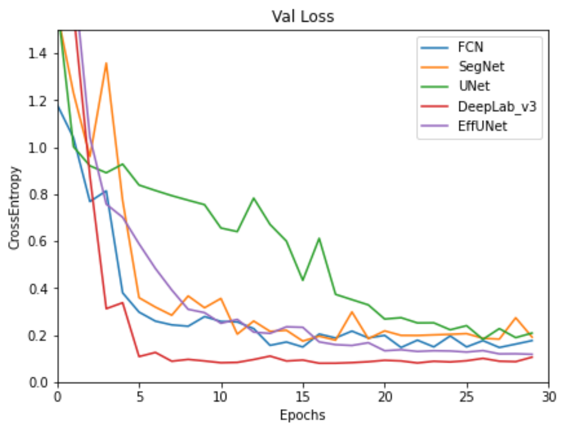
FCN(Blue Curve) is overfitting on the training set. DeeplabV3+(Green Curve) has the lowest validation error and intermediate training loss. EffUNet(Purple Curve) and DeeplabV3+ are similar except that EffUNet sometimes has unstable validation loss. The models from the Transformer family(Brown and Pink Curve) haven't performed normally on this dataset. And we believe the most important reason is the limited amount of data. As DeepLabV3+ shows the most stable and promising results, it is chosen for this problem.

3.3 Ensemble

In [ ]:

```
# Num epochs
EPOCHS = 30

def get_config_model(model):
    """Gets the configuration dictionary.

    Args:
        model (dict): The model description

    Returns:
        (dict): Full config dictionary on which the training depends
    """
    # Describe the optimizer
    optimizer = {
        "name": "Adam",
        "lr": 0.001,
        "weight_decay": 1e-7
    }

    return get_config(model, optimizer, epochs=EPOCHS, batch_size=32, loss="CrossEntropy")

# Define possible model choices
MODEL_CHOICES = [
    {"name": "FCN", "input_channels": 3, "model_path": "models/fcn_model.pth"},
    {"name": "SegNet", "input_channels": 3, "model_path": "models/segnet_model.pth"},
    {"name": "UNet", "depths": 4, "input_channels": 1, "model_path": "models/unet_model.pth"},
    {"name": "DeepLab_v3", "backbone": "drn", "input_channels": 3, "model_path": "models/deeplab
_model.pth"},
    {"name": "EffUNet", "input_channels": 3, "model_path": "models/effunet_model.pth"}
]

# Define the labels
MODEL_LABELS = [model["name"] for model in MODEL_CHOICES]

# Plot the results
individual_models, _, _ = plot_comparisons(get_config_model, MODEL_CHOICES, MODEL_LABELS, loss_n
ame="CrossEntropy")
```





In [ ]:

```
# Initialize empty configs
configs = []

for i, model in enumerate(individual_models):
    # Get config and save the model
    config = get_config_model(MODEL_CHOICES[i])
    save_model(model, config)
    configs.append(config)

    # Also evaluate each model's accuracy
    print(config["model_info"]["name"] + " performance:")
    eval(config)

# Define ensemble model info
ENSEMBLE = {
    "name": "Ensemble",
    "input_channels": 1,
    "configs": configs,
    "model_path": "models/ensemble_model.pth"
}

# Get ensemble config
ensemble_config = get_config_model(ENSEMBLE)

# Evaluate the total performance
print("Ensemble performance:")
eval(ensemble_config, ensemble=True)
```

```
FCN performance:
Average dice loss on the validation set = 0.8347135782241821
SegNet performance:
Average dice loss on the validation set = 0.722546398639679
UNet performance:
Average dice loss on the validation set = 0.7176639437675476
DeepLab_v3 performance:
Average dice loss on the validation set = 0.8510346412658691
EffUNet performance:
Average dice loss on the validation set = 0.8352847099304199
Ensemble performance:
Average dice loss on the validation set = 0.872326672077179
```

As shown from the above, the ensemble architecture has shown the highest accuracy (the average dice loss above refers to accuracy) out of all the above architectures, which is 2% higher than the second best model, DeepLabV3+. However, as it has not been thoroughly tested and tuned yet it is not used as the focus of the report.

4 Further Improvements

In 2.2.7, it can be seen that a simple version of data augmentation, randomised flip and rotation, is utilised. The augmentation didn't change the number of images, but focused on variability given the limited data. Besides, methods such as transpose and blur are performing badly, which is presumed to be the fact that there are no such complicated samples in test data. Although current data augmentation didn't bring an obvious improvement (~1%), further improvements will be mainly focusing on more complicated augmentation techniques since the limitation of data has become the prime reason for models, in particular those with a larger number of parameters.

Additionally, contrastive learning, an emerging technique in self-supervised learning, might be of significance in semantic segmentation. The core idea of contrastive learning is to represent samples in a more discriminative feature space, so that samples projected to the space can be easily recognised. Its theory is similar to AutoEncoder (AE), though AE will reconstruct an entire sample whereas a contrastive learning based model will only learn a space that is discriminative enough.

5 Conclusion

This final section provides a summary on the results achieved doing the coursework. Most of the members in the team have contributed and spent significant amounts of time researching and implementing the various architectures, and gained insights and understanding of the given task, semantic medical image segmentation. The DeepLabV3+ model with DRN backbone is chosen as the final, best model from all the tested models after careful consideration and evaluation. After numerous experiments, it is found out that the model architecture, hyperparameters, including number of epochs run, learning rate, type of backbone, the batch size, the optimiser, and the loss function used all affects the performance of the model. Among them, the batch size, optimiser and the loss function showed significant difference in performance before and after experimenting and discovering the optimal choice for them (As discussed in section 3). If there were more data samples available, it is possible to achieve a higher accuracy as the model will be able to see more data and will be more generalised. The ensemble structure was discovered and developed at a later phase, as it is not fully developed and tuned, it is decided that the report will still focus on the DeepLabV3+ structure. With the limited data samples, the best generalisation performance achieved by submission to the Kaggle platform has an 86.096% accuracy, which was satisfactory.

References

[1]: ACDC challenge and the dataset, https://www.creatis.insa-lyon.fr/Challenge/acdc/ (https://www.creatis.insa-lyon.fr/Challenge/acdc/).

[2]: Dice similarity coefficient, https://en.wikipedia.org/wiki/Sørensen–Dice-coefficient (https://en.wikipedia.org/wiki/Sørensen–Dice-coefficient).

[3]: L. Najman and M. Schmitt, “Watershed of a continuous function”, Signal Processing, vol. 38, no. 1, pp. 99–112, 1994.

[4]: R. Nock and F. Nielsen, “Statistical region merging”, IEEE Transactions on pattern analysis and machine intelligence, vol. 26, no. 11, pp. 1452–1458, 2004.

[5]: Hangbo Bao, Li Dong, Furu Wei, “BEiT: BERT Pre-Training of Image Transformers”, 2021.

[6]: Ze Liu, Han Hu, Yutong Lin, Zhuliang Yao, Zhenda Xie, Yixuan Wei, Jia Ning, Yue Cao, Zheng Zhang, Li Dong, Furu Wei, Baining Guo, “Swin Transformer V2: Scaling Up Capacity and Resolution”, 2021.

[7]: Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, Demetri Terzopoulos, “Image Segmentation Using Deep Learning: A Survey”, 2020.

[8]: Jonathan Long, Evan Shelhamer, Trevor Darrell, “Fully Convolutional Networks for Semantic Segmentation”, 2014.

[9]: Vijay Badrinarayanan, Alex Kendall, Roberto Cipolla, “SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation”, 2015.

[10]: Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, Alan L. Yuille, “DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs”, 2016.

[11]: Barret Zoph, Golnaz Ghiasi, Tsung-Yi Lin, Yin Cui, Hanxiao Liu, Ekin D. Cubuk, Quoc V. Le, “Rethinking Pre-training and Self-training”, 2020.

[12]: Gabriele Valvano, Andrea Leo, Sotirios A. Tsaftaris, “Learning to Segment from Scribbles using Multi-scale Adversarial Attention Gates”, 2021.

[13]: Gabriele Valvano, Andrea Leo, Sotirios A. Tsaftaris, “Self-supervised Multi-scale Consistency for Weakly Supervised Segmentation Learning”, 2021.

[14]: O. Ronneberger, P. Fischer, and T. Brox, “U-net: Convolutional networks for biomedical image segmentation,” in International Conference on Medical image computing

[15]: L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, ‘Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs’, ArXiv14127062 Cs, Jun. 2016, Accessed: Nov. 04, 2021. [Online]. Available: http://arxiv.org/abs/1412.7062 (http://arxiv.org/abs/1412.7062)

[16]: L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, ‘DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs’, ArXiv160600915 Cs, May 2017, Accessed: Nov. 25, 2021. [Online]. Available: http://arxiv.org/abs/1606.00915 (http://arxiv.org/abs/1606.00915)

[17]: L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, ‘Rethinking Atrous Convolution for Semantic Image Segmentation’, ArXiv170605587 Cs, Dec. 2017, Accessed: Nov. 25, 2021. [Online]. Available: http://arxiv.org/abs/1706.05587 (http://arxiv.org/abs/1706.05587)

[18]: L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, ‘Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation’, ArXiv180202611 Cs, Aug. 2018, Accessed: Nov. 06, 2021. [Online]. Available: http://arxiv.org/abs/1802.02611 (http://arxiv.org/abs/1802.02611)

[19]: U-Net: Convolutional Networks for Biomedical Image Segmentation, https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/ (https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/)

[20]: F. Yu, V. Koltun, and T. Funkhouser, ‘Dilated Residual Networks’, ArXiv170509914 Cs, May 2017, Accessed: Nov. 26, 2021. [Online]. Available: http://arxiv.org/abs/1705.09914 (http://arxiv.org/abs/1705.09914)

[21]: Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems. 2017.

[22]: Zheng, Sixiao, et al. "Rethinking semantic segmentation from a sequence-to-sequence perspective with transformers." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2021.

[23]: Baheti, Bhakti, et al. "Eff-unet: A novel architecture for semantic segmentation in unstructured environment." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition