Coursework II 2020/2021 Report

Cardiovascular MR Segmentation

Group 8

A report for the semantic segmentation of magnetic resonance (MR) images during the coursework of the Neural Computation module, 2020

Project By:

Silin Li - SXL1046@student.bham.ac.uk
Yiwen Ge - YXG883@student.bham.ac.uk
Zhen Su - ZXS777@student.bham.ac.uk
Kunyong Li- KXL706@student.bham.ac.uk
Yueyi Wang - YXW835@student.bham.ac.uk

UNIVERSITY OF BIRMINGHAM

College of Enginnering & Physical Sciences
School of Computer Science

1. Introduction

1.1 Dataset

Our models are trained end-to-end from scratch using ACDC1 challenge dataset which is modified for the sake of simplicity. The dataset contains a total of 200 CMR images in a PNG format. And the mask image is a grayscale image (8-bit depth) of 96 by 96 pixels. Out of these 100 are used for training phase, 80 for testing phase and 20 for validation phase of this challenge.

1.2 Task

Introduction MR is a biological magnetic spin imaging technology, which uses the characteristics of the spin motion of atomic nuclei to generate a signal after RF pulse excitation in an external magnetic field, which is detected by a detector and input to a computer, and the image is displayed on the screen after processing. Cardiac MR images can generate very detailed images of the left ventricle, the right ventricle and the myocardium in the heart. The depiction of these structures can provide relevant diagnostic information and evaluate the overall function of the heart. Because it may take a long time to collect these large amounts of image data, and the hand-drawn image will cause some errors and accuracy of details, and the patient's condition cannot be well-positioned and evaluated. So, we need an accurate and time-consuming method of heart segmentation. Recently, in medical image analysis, the deep learning convolutional neural network CNN has a very significant effect, but it usually requires a large training data set and provides sub-optimal results that need further improvement. Here, we developed a method to enhance cardiac MRI multi-class segmentation by combining the advantages of CNN and interpretable machine learning algorithms.

1.3 Aim

For this reason, we modified the u-net model to extract elements of different spatial scales and then up-sampled and increased all the spatial resolution of the learning elements finally achieves a high separation accuracy. In this experiment, we found the MR data set from the open source website and fully analyzed its test set, training set and validation set. The goal is to design a neural network and improve the efficiency of image analysis and the accuracy of the image, and it can be close to the accuracy of the real image. 90% of the rate. Through the use of smaller training data sets and improved segmentation accuracy and repeatability of cardiac MRI segmentation in research and clinical patient care.

2. Implementation

2.1 Neural Network Description and Justification

Semantic Segmentation is an important part of image processing and image understanding in machine vision technology, and also an important branch of AI field. At present, Convolutional Neural Network(CNN) has made great achievements in image classification. Network structures such as VGG and Resnet have emerged and achieved great results in ImageNet. The different between semantic segmentation and classification is that the former requires the classification of each pixel in the image to be accurately segmented. However, image details are lost in convolution and pooling conducted by Convolutional Neural Network(CNN). It means that the feature map size gradually decreases. Therefore, it is not possible to indicate the specific outline of the object and the object to which each pixel belongs, and accurate segmentation cannot be achieved.

2.1.1 Simple CNN model

CNN is one of the powerful deep learning neural networks at present, it has a amazing performance on image-based classification due to CNN's special stucture of weight sharing. This special structure of weight sharing significantly reduce the complexity of the network, and has a better ability of generalization compares to other type of neural networks. In the coursework each cardiovascular MR image is segmented into a mask image, which has four different regions. It can be converted to the classification problem that each pixel on the cardiovascular MR image belongs to the one of the four regions. CNN's architecture is composed of multiple layers, each of which performs a specific function and translates it into a usable representation. Among them, in complex neural networks, we can usually find three main types of layers. The first is the Convolutional layer. It can also be called the transformation layer and forms the foundation of CNN. It is the core operation of firing network neurons. And followed by ReLu (Rectified Linear Unit) Layer, is the most commonly used output neurons (CNN) activation function. Finally, the pooling layer is mainly used to reduce the spatial dimension of the input volume for the next convolutional layer.[2]

2.1.2 U-net

The network architecture is illustrated in Figure 1. And it looks like a "U" which justifies its name. It consists of a contracting-path which is left side and an expansive-path which is left side. U-net is a typical Encoder-Decoder structure. Encoder for feature extraction or down sampling, decoder for up-sampling. More sepcificlly, the left encoder part is used to gradually display the environmental information. The right decoder part is used to combine the down sampling information of each layer and up-sampling input information to restore the detailed information. In the middle is skip-connect for feature fusion. Thus U-Net combines the location information from the down sampling to obtain a general information combining localization and context, which is necessary to make a segmentation on cardiovascular MR images. All the convolutional layers in the network are 3×3 convolution except the final output layer. At the final layer a 1×1 convolution is used to map each feature vector to the desired number of classes.[1]

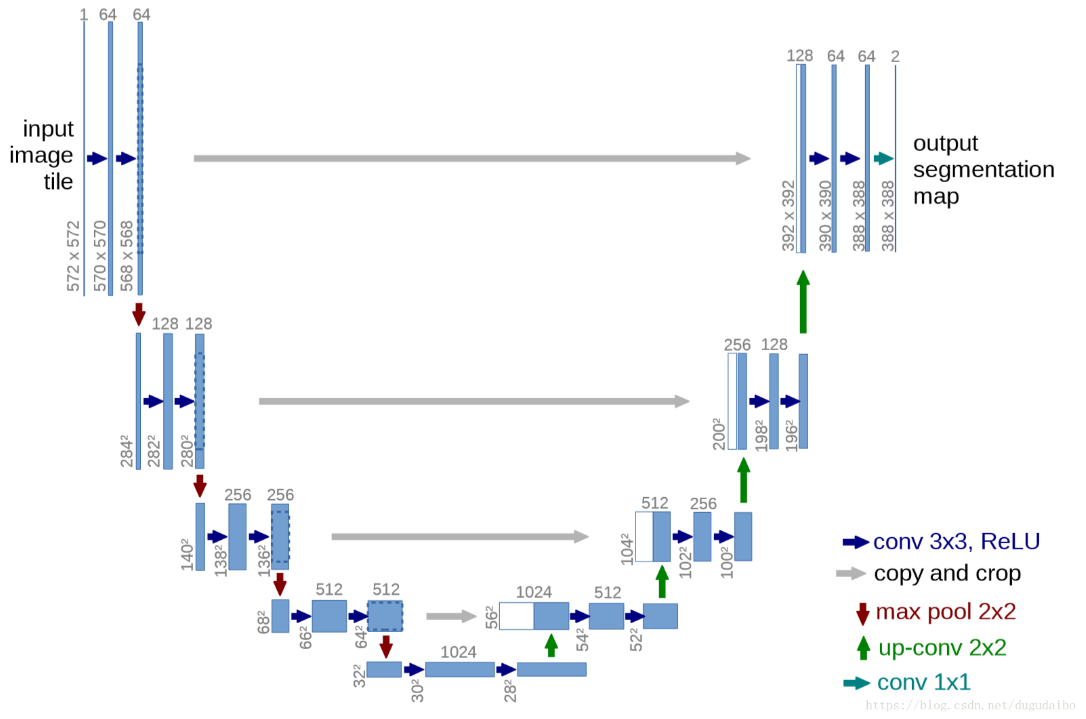

Figure 1 : U-net architecture

2.2 Implementation

2.2.1 Image preprocess

In the beginning, we apply the given 100 pairs of image and mask pictures to train our model. The training effect of the model looks good. The average accuracy of the validation set can reach approximately 0.885. After a certain training, the accuracy of the validation set even reached 0.896. However, we should be aware of a problem, the number of pictures in the training set may not be sufficient.

When we train a machine learning model, what you're really doing is tuning its parameters such that it can map a particular input (say, an image) to some output (a mask).[6] For a picture, we see some objects, and for the machine, what it sees is some pixels. If we photograph the same object, and we change the position of the camera. The picture may change very little for our human eyes, but the picture pixels change very much for the machine. There are many other elements that also affect machine learning. For example, the degree of deformation of the object in the image, the brightness of image, the shelter in the image. Therefore, the more the number of samples in the training set, the better the effect of the trained model. Increasing the amount of training data will also improve the generalization ability of the model. In addition, increasing the data with some noise is helpful to improve the robustness of the model. [7]

Then we will introduce how we implemented data enhancement to efficiently feed the input data volumes into our model. We took advantage of a third-party library of data enhancements called Augmentor.[8] We call Augmentor.Pipeline() to put all the original images into pipeline and call p.ground_truth() point to a directory containing original mask data. Images with the same file names will be added as mask data and augmented in parallel to the original data. The rotate() function is used to rotate pictures, the flip_left_right() function is used to flip the picture left and right and the flip_top_bottom() function is used to flip the picture up and down. And each function can set the probability in order to randomly perform various data enhancement operations on the original image. We can call the sample() function to set the final number of pairs of image and mask after data enhancement we want. So, to get more data, we just need to make minor alterations to our existing dataset. Minor changes such as flips or translations or rotations. Our neural network would think these are distinct images anyway.[6]

2.2.2 Training

Initially, we constructed a basic U-net using 5 downsampling layers and 5 up sampling layers. Each downsampling layer has consisted of 2 convolutional layers followed by a Batch normalization layer, ReLU activation. Each upsampling layer is composed of an up-sampling convolutional layer a feature concatenation layer and two 3x3 convolutional layers (ReLU) repeatedly. The default kernel size for all the convolutional layers is set to be 3 *3 except the last layer. The last convolutional layer is applied with a kernel size of 1* 1 to reduce the channels of the segmentation mask to 4.

U-net starts with 96 *96 CMR images of a single channel. When the image passes through the 5 downsampling layers, the size of the image will be reduced by twice until it is compressed into 6* 6 space in the final stage of encoding. The depth of the model can be controlled by the number of pooling layers, so we hope the model goes deeper that can capture more simple information of the image through down-sampling, and get more deep and abstract features through up-sampling. Thus we decided to halve the image in each encoding layer. For the channels of the kernel, we have tried to gradually increase the maximum number of kernel channels at the down sampling layers to 384(96 *4), 768(96* 8), 1536(96 *16), and 1920(96* 20) the result showed that although a large amount of parameter of maximum kernel channels of 1536 and lower running speed, it has a better performance on the Dice score of all class. The figure2 show the performance of different model on the average of Dice score of 3 classes.

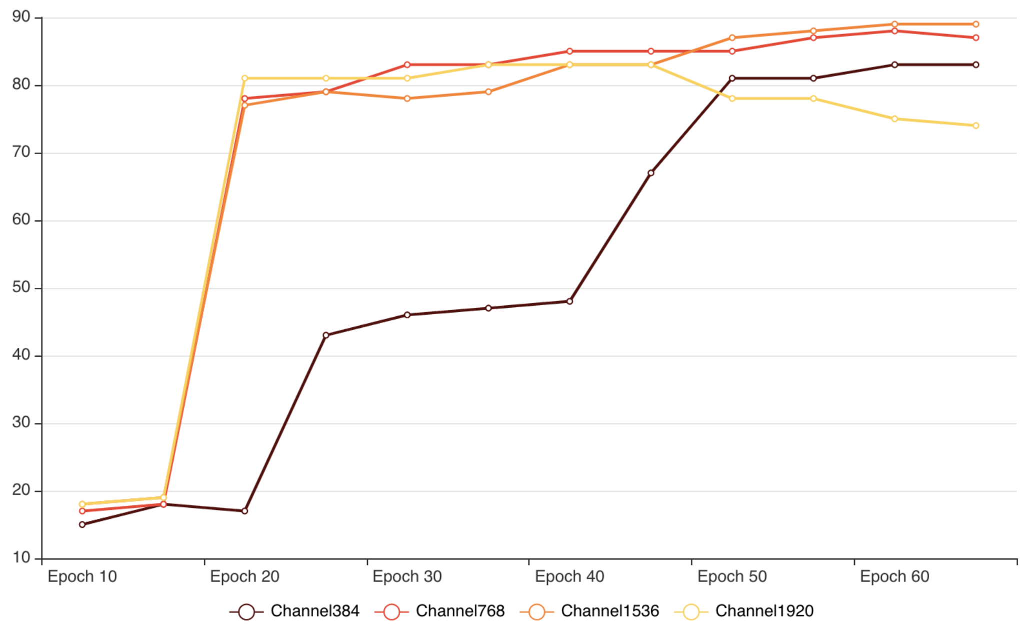

Figure 2 : Model Comparison

We chose transposed convolution as the main method of upsampling. It was able to restore the original size of the feature map, which had been shrunk after the convolution operation. Since the larger size feature map obtained by upsampling lacks information, and some edge features will be lost every time the feature is extracted by downsampling. Concatenation of shallow feature maps with deep feature maps can retrieve some of the lost features.

The figue below is the structure of our U-net model, the source code is in the end of paper.

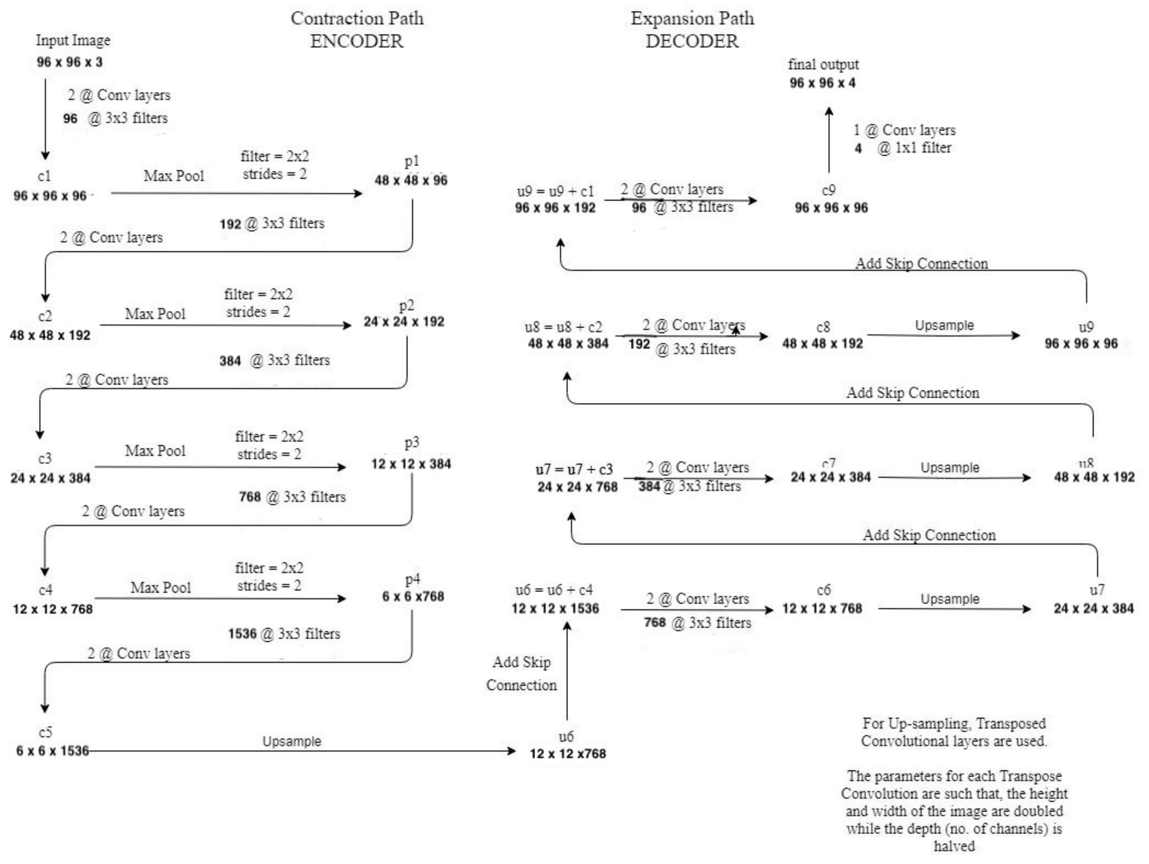

Figure 3 : U-net structure

2.2.3 Inference

For the inference, we used a strategy that save the best model with the lowest loss in the training process instead of saving the model after the last epoch. Because it is possible that the performance of model in the last epoch worse than the model in other epoch. Although a lower loss value does not represents better generalization ability and higher accuracy, it is still an important criterion for evaluating the model. In fact, any epoch after the loss converges may be the optimal model. Instead of only saving the model of the last epoch, it is better to use loss as the criterion for saving. To solve this problem, we decided to save best model(best_model.pth) with the lowest loss and last model(last_model.pth) in the last epoch and evaluate the performance of these two models on test and valiadation set, then keep the model with the highest accuracy.

3. Experimentation

For our task, we have designed our models based on U-net arcitecture, and the structure of our model is fixed. So, in this experiment section, we tested combinations on the parameters of different optimizers and loss functions. For each set of combination, we tuned the hyperparameters including epochs, learning rate, momentum to accelerate training speed and weight decay to avoid overfitting. We used train losses and validation accuracy rates as the reference to choose the parameteres. Due to the large amount of data in our model, we decided to used GPU to train this model. So we run all experiments on the GistGPU workstation equipped with 14 GB of memory, Intel(R) Core(TM) with a 11 GB NVidia 1070Ti GPU.

3.1 Optimizer and loss function

For CNN-like architectures, we found 3 optimizers that coders used the most, which is Adm, RNSprop, and SGD(+Nesterov). The Adam is considered as efficient to implement and suitable for a large dataset with many feature parameters. From the Adam: A Method of Stochastic Optimization [3], we saw the outstanding performance it presented on the CIFAR-10 dataset compared to the other optimizers. During reading this paper, we found SGD is compatible with Adam, and we picked RNSprop as a comparison to its upgrading Adam. We expect Adam to show the best results and how much SGD should perform compared to Adam.

For loss function, we referred to those popular loss functions on image classification and tested on CrossEntropyLoss, BinaryCrossEntropyLoss, BCEwithlogitsLoss, DiceLoss[4], and FocalLoss[5]. In our research on loss functions, we expected dice loss and cross-entropy loss to perform well as they are commonly used to classify many classes and measure pixels. BCE loss creates a criterion on predicted image and mask. Focal loss seems to be suitable for binary classification but could be refined using the tuning of parameter gamma.

We used to torch predefined loss functions and referred to the implementations on github.com. The arithmetic models of the dice loss layer and focal loss layer are from reference papers. The experimenting process is we train on one set of Optimizer and Loss, and when train losses for one epoch are minimized, we record the weight.pth for validation. Finally, we use validation data to calculate the model accuracy rate with the categorical dice criterion.

With test results from training and evaluation, the criterion for the model is the average predicted accuracy on the validation dataset. Also, considering the optimizers are easily affected by the hyperparameters, we picked the best sets of both Adam and SGD for later tests.

3.2 Hyperparameter

In our model, we have two sets of hyperparameters requires tuning.

One set is for training model, {epoch, batch size}; and the other one is for optimizer, {batch size, learning rate(lr), momentum(m), weight decay rate(wd)}. Specifically, we tested [Adam,lr] and SGD [lr, m ,wd,].

In the tests, we controlled the other varieties and test one hyper parameter per test. The validation criterion is as same as the optimizer/loss part.

sgd-focal	sgd-bcewithlogits	rms-focal	adam-focal	rms-celoss	adam-bcewithlogit	sgd-bce	rms-softdi
0	0.231340435	0.620937215	0.816520201	0.832665331	0.834759565	0.851704171	0.85611

rms - soft dice	rms-bcewithlogits	adam-softdice	sgd-softdice	rms-bce	adam-bceloss	adam-celoss	
0.85611613	0.856642402	0.862051213	0.866925476	0.877351317	0.888750147	0.892168325	0

Table 1 : {optimizer-Loss: Mean accuracy on validation dataset in Ascending order}

3.3 Conclusion

From the experiments, we finally decided to use {Optimizer: Adam; Loss Function: CrossEntropy } and the hyperparameters is {batch size:4, learning rate 0.001, epoch:60}. (we also saved the best case for SGD {batch size:4, learning rate: 0.01, epoch:60, momentum: 0.9, weight decay: 0.0001}, for SGD descends slowly we increase the learning rate and adds weight decay so that SGD does not overfit)

adam-celoss	**epoch = 60**	**epoch = 70**
Ave-acc	0.892168325	0.8380

Table 2 : {Hyper parameters: Normed Average accuracy on validation dataset in Ascending order}

Notably, Adam shows general adaptation for all the loss function and performed accuracy of 85.4% in average with standard deviation of 0.3212. While SGD is affected easily by loss function choice as the accuracy of sgd-focal is 0.0, so despite SGD even out wins Adam from Table 1, we still decided using Adam. We discussed why this would happen and concluded that SGD used a larger learning rate, so the optimizer is quick to fall into a partial optimization point and this whether this point is the best case differs from every test. On the other hand, Adam can record the $momentum^2$ descent and $gradient^2$ descent which makes Adam more adaptive than the other.

About SGD+N and SGD: Every paper and tutorial we found displays the refinement SGD+N has made over SGD, but in our test, SGD performs better than SGD+N.

	diceloss	**bce**	**ce**
SGD	0.866925476	0.851704171	0.895455727
SGD+N	0.8132	0.7872	0.7736

Table 3 : {SGD/SGD-N: Mean accuracy on validation dataset}

4. Conclusion

In this paper, we implemented a U-net based neural network to make a segmentation of CMR images. We found from the experimentation that the Dice score of first class in the mask is lower than the other two classes except for forth class(background). For the results obtained from U-Net, we conclude that a robust and better generalization U-Net model with more channels and pooling layers shows better performance. Having implemented multiple tests on the combination of optimizer and loss function and tuned the hyperparameter, we experienced the remarkable performance Adam could provide as one integrated optimizer featuring gradient decay and momentum decay. As we tuned the hyperparameters of SGD, we experienced the evaluation of optimizers from simple gradient decay to the stage with momentum decay and finally to the stage with weighted gradient decay. Or we could say, the hyperparameters in SGD displayed the progress in the refinement of optimizers. Overall, our model has achieved good results in kaggle, 89.641, but there are still areas that need improvement, such as trying other network structures.

Reference

- [1] Olaf Ronneberger, Philipp Fischer, & Thomas Brox. U-net: Convolutional networks forbiomedical image segmentation. InInternational Conference on Medical image computingand computer-assisted intervention. Springer, 2015.
- [2] Saxena, A. (2016, June 29). Convolutional Neural Networks (CNNs): An Illustrated Explanation. Retrieved December 16, 2020, from https://blog.xrds.acm.org/2016/06/convolutional-neural-networks-cnns-illustrated-explanation/.
- [3] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, Nov. 1998, doi: 10.1109/5.726791.
- [4] Hubutui. (n.d.). Hubutui/DiceLoss-PyTorch. Retrieved December 17, 2020, from https://github.com/hubutui/DiceLoss-PyTorch/blob/master/loss.py
- [5] CoinCheung. (n.d.). CoinCheung/pytorch-loss. Retrieved December 17, 2020, from https://github.com/CoinCheung/pytorch-loss/blob/master/focal_loss.py
- [6] Gandhi, A. (2019, August 06). Data Augmentation: How to use Deep Learning when you have Limited Data. Retrieved December 17, 2020, from https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/
- [7] Shorten, C., Khoshgoftaar, T.M. A survey on Image Data Augmentation for Deep Learning. J Big Data 6, 60 (2019). https://doi.org/10.1186/s40537-019-0197-0
- [8] Mdbloice. (n.d.). Mdbloice/Augmentor. Retrieved December 17, 2020, from https://github.com/mdbloice/Augmentor

---

Source Code

Image augmentation

In [ ]:
```
#Data_enhance.py

import Augmentor

#Hint:the name of png format file in the original image folder should keep sa
#the name of png format file in the original mask folder. Otherwise, Augmento
#find the corresponding mask image. And it will not generate a one-to-one cor
#image file and mask file
p = Augmentor.Pipeline("data/train/image_original")
# Point to a directory containing ground truth data.
# Images with the same file names will be added as ground truth data
# and augmented in parallel to the original data.
p.ground_truth("data/train/mask_original")
# Add operations to the pipeline as normal:
p.rotate(probability=1, max_left_rotation=5, max_right_rotation=5)
p.flip_left_right(probability=0.5)
p.flip_top_bottom(probability=0.5)
p.sample(200)
```

In [ ]:
```
#Data_update_subscript.py

import os
import shutil

def dispatch():
    root_dir = 'data/train/image_original/output'
    image_new = 'data/train/image'
    mask_new = 'data/train/mask'
    if not os.path.exists(image_new):
        os.makedirs(image_new)
    if not os.path.exists(mask_new):
        os.makedirs(mask_new)



    files = os.listdir(root_dir)
    count = 1
    for filename in files:
        if filename.startswith('_groundtruth'):
            mask_path = os.path.join(root_dir, filename)
            # count-x, set x everytime, it is corresponding to the total numb
            shutil.copyfile(mask_path, os.path.join(mask_new, 'cmr' + '%d_mas
            count +=1

        elif filename.startswith('image_original_'):
            image_path = os.path.join(root_dir, filename)
            shutil.copyfile(image_path, os.path.join(image_new, 'cmr' + '%d.p
            count +=1
    print(count-1)


dispatch()
```

Load data from gitlab

Define a DataLoader

In [ ]:
```
import torch
import torch.utils.data as data
import cv2
import os
from glob import glob

class TrainDataset(data.Dataset):
    def __init__(self, root=''):
        super(TrainDataset, self).__init__()
        self.img_files = glob(os.path.join(root,'image','*.png'))
        self.mask_files = []
        for img_path in self.img_files:
            basename = os.path.basename(img_path)
            self.mask_files.append(os.path.join(root,'mask',basename[:-4]+'_m


    def __getitem__(self, index):
            img_path = self.img_files[index]
            mask_path = self.mask_files[index]
            data = cv2.imread(img_path, cv2.IMREAD_UNCHANGED)
            label = cv2.imread(mask_path, cv2.IMREAD_UNCHANGED)
            return torch.from_numpy(data).float(), torch.from_numpy(label).fl

    def __len__(self):
        return len(self.img_files)

class TestDataset(data.Dataset):
    def __init__(self, root=''):
        super(TestDataset, self).__init__()
        self.img_files = glob(os.path.join(root,'image','*.png'))
    def __getitem__(self, index):
            img_path = self.img_files[index]
            data = cv2.imread(img_path, cv2.IMREAD_UNCHANGED)
            return torch.from_numpy(data).float()

    def __len__(self):
        return len(self.img_files)
```

Define a Segmenatation Model

In [ ]:
```
import torch
from torch import nn


class CNNblock(nn.Module):
    def __init__(self, in_channels, features):
        super(CNNblock, self).__init__()
        self.double_conv = nn.Sequential(
            nn.Conv2d(in_channels=in_channels,out_channels=features,kernel_si
            nn.BatchNorm2d(num_features=features),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=features,out_channels=features,kernel_size=
            nn.BatchNorm2d(num_features=features),
            nn.ReLU(inplace=True),
        )

    def forward(self, x):
        return self.double_conv(x)


class CNNSEG(nn.Module):

    def __init__(self, in_channels=1, out_channels=4, features_size=96, debug
        super(CNNSEG, self).__init__()

        features = features_size
        self.debug = debug
        self.encoder1 = CNNblock(in_channels, features)#[1,96,96,96]
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.encoder2 = CNNblock(features, features*2)#[1,192,48,48]
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.encoder3 = CNNblock(features*2, features*4)#[1,384,24,24]
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.encoder4 = CNNblock(features*4, features*8)#[1,768,12,12]
        self.pool4 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.bottleneck = CNNblock(features*8, features*16)#[1,1536,6,6]

        self.upconv4 = nn.ConvTranspose2d(features*16, features*8, kernel_siz
        self.decoder4 = CNNblock(features*16, features*8)#[1,768,12,12]
        self.upconv3 = nn.ConvTranspose2d(features*8, features*4, kernel_size
        self.decoder3 = CNNblock(features*8, features*4)#[1,384,24,24]
        self.upconv2 = nn.ConvTranspose2d(features*4, features*2, kernel_size
        self.decoder2 = CNNblock(features*4, features*2)#[1,192,48,48]
        self.upconv1 = nn.ConvTranspose2d(features*2, features, kernel_size=2
        self.decoder1 = CNNblock(features*2, features) #[1,96,96,96]

        self.conv = nn.Conv2d(in_channels=features, out_channels=out_channels
                                                        #[1,4,96,96]

    def forward(self, x):
        enc1 = self.encoder1(x)
        pool1 = self.pool1(enc1)
        enc2 = self.encoder2(pool1)
        pool2 = self.pool2(enc2)
        enc3 = self.encoder3(pool2)
        pool3 = self.pool3(enc3)
        enc4 = self.encoder4(pool3)
        pool4 = self.pool4(enc4)

        bottleneck = self.bottleneck(pool4)

        dec4 = self.upconv4(bottleneck)
        dec4 = self.decoder4(torch.cat((dec4, enc4), dim=1))
        dec3 = self.upconv3(dec4)
        dec3 = self.decoder3(torch.cat((dec3, enc3), dim=1))
        dec2 = self.upconv2(dec3)
        dec2 = self.decoder2(torch.cat((dec2, enc2), dim=1))
        dec1 = self.upconv1(dec2)
        dec1 = self.decoder1(torch.cat((dec1, enc1), dim=1))

        if self.debug:
            print('encode1:', enc1.size())
            print('encode2:', enc2.size())
            print('encode3:', enc3.size())
            print('encode4:', enc4.size())
            print('bottleneck:', bottleneck.size())
            print('decode4:', dec4.size())
            print('decode3:', dec3.size())
            print('decode2:', dec2.size())
            print('decode1:', dec1.size())
            print(self.conv(dec1).size())

        return torch.sigmoid(self.conv(dec1))

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = CNNSEG().to(device)

print(device)
print(model)
```

Other Loss we test

In [ ]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np

# from https://github.com/hubutui/DiceLoss-PyTorch/blob/master/loss.py
def make_one_hot(input, num_classes):
    """Convert class index tensor to one hot encoding tensor.
    Args:
         input: A tensor of shape [N, 1, *]
         num_classes: An int of number of class
    Returns:
        A tensor of shape [N, num_classes, *]
    """
    shape = np.array(input.shape)
    shape[1] = num_classes
    shape = tuple(shape)
    result = torch.zeros(shape)
    result = result.scatter_(1, input.cpu(), 1)

    return result


class BinaryDiceLoss(nn.Module):
    """Dice loss of binary class
    Args:
        smooth: A float number to smooth loss, and avoid NaN error, default:
        p: Denominator value: \sum{x^p} + \sum{y^p}, default: 2
        predict: A tensor of shape [N, *]
        target: A tensor of shape same with predict
        reduction: Reduction method to apply, return mean over batch if 'mean
            return sum if 'sum', return a tensor of shape [N,] if 'none'
    Returns:
        Loss tensor according to arg reduction
    Raise:
        Exception if unexpected reduction
    """
    def __init__(self, smooth=1, p=2, reduction='mean'):
        super(BinaryDiceLoss, self).__init__()
        self.smooth = smooth
        self.p = p
        self.reduction = reduction

    def forward(self, predict, target):
        assert predict.shape[0] == target.shape[0], "predict & target batch s
        predict = predict.contiguous().view(predict.shape[0], -1)
        target = target.contiguous().view(target.shape[0], -1)

        num = torch.sum(torch.mul(predict, target), dim=1) + self.smooth
        den = torch.sum(predict.pow(self.p) + target.pow(self.p), dim=1) + se

        loss = 1 - num / den

        if self.reduction == 'mean':
            return loss.mean()
        elif self.reduction == 'sum':
            return loss.sum()
        elif self.reduction == 'none':
            return loss
        else:
            raise Exception('Unexpected reduction {}'.format(self.reduction))


class DiceLoss(nn.Module):
    """Dice loss, need one hot encode input
    Args:
        weight: An array of shape [num_classes,]
        ignore_index: class index to ignore
        predict: A tensor of shape [N, C, *]
        target: A tensor of same shape with predict
        other args pass to BinaryDiceLoss
    Return:
        same as BinaryDiceLoss
    """
    def __init__(self, weight=None, ignore_index=None, **kwargs):
        super(DiceLoss, self).__init__()
        self.kwargs = kwargs
        self.weight = weight
        self.ignore_index = ignore_index

    def forward(self, predict, target):
        assert predict.shape == target.shape, 'predict & target shape do not 
        dice = BinaryDiceLoss(**self.kwargs)
        total_loss = 0
        predict = F.softmax(predict, dim=1)

        for i in range(target.shape[1]):
            if i != self.ignore_index:
                dice_loss = dice(predict[:, i], target[:, i])
                if self.weight is not None:
                    assert self.weight.shape[0] == target.shape[1], \
                        'Expect weight shape [{}], get[{}]'.format(target.sha
                    dice_loss *= self.weights[i]
                total_loss += dice_loss

        return total_loss/target.shape[1]

# from https://github.com/CoinCheung/pytorch-loss/blob/master/focal_loss.py
# version 1: use torch.autograd
class FocalLossV1(nn.Module):

    def __init__(self,
                 alpha=0.25,
                 gamma=2,
                 reduction='mean',):
        super(FocalLossV1, self).__init__()
        self.alpha = alpha
        self.gamma = gamma
        self.reduction = reduction
        self.crit = nn.BCEWithLogitsLoss(reduction='none')

    def forward(self, logits, label):
        '''
        logits and label have same shape, and label data type is long
        args:
            logits: tensor of shape (N, ...)
            label: tensor of shape(N, ...)
        '''

        # compute loss
        logits = logits.float() # use fp32 if logits is fp16
        with torch.no_grad():
            alpha = torch.empty_like(logits).fill_(1 - self.alpha)
            alpha[label == 1] = self.alpha

        probs = torch.sigmoid(logits)
        pt = torch.where(label == 1, probs, 1 - probs)
        ce_loss = self.crit(logits, label.float())
        loss = (alpha * torch.pow(1 - pt, self.gamma) * ce_loss)
        if self.reduction == 'mean':
            loss = loss.mean()
        if self.reduction == 'sum':
            loss = loss.sum()
        return loss
```

Define a Loss function and optimizer

In [ ]:

```
Loss = nn.CrossEntropyLoss()
# Loss = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
# optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

Training

In [ ]:

```
from torch.utils.data import DataLoader
import numpy as np
import time
from matplotlib import pyplot as plt

def convert_to_4_channel(img):
    converted_img = torch.zeros((img.shape[0], 4, img.shape[1], img.shape[2])
    # Binarization
    converted_img[:, 0, :, :] = torch.where(img == 0, 1, 0)
    converted_img[:, 1, :, :] = torch.where(img == 1, 1, 0)
    converted_img[:, 2, :, :] = torch.where(img == 2, 1, 0)
    converted_img[:, 3, :, :] = torch.where(img == 3, 1, 0)
    return converted_img

#################################################### Load dataset#############

train_data_path = './data/train'
val_data_path = './data/val'
num_workers = 4
batch_size = 10
train_set = TrainDataset(train_data_path)
training_data_loader = DataLoader(dataset=train_set, num_workers=0, batch_siz
val_set = TrainDataset(val_data_path)
val_data_loader = DataLoader(dataset=val_set, num_workers=num_workers, batch_

###################################################### Training #############
train_loss=[]
loss_training_epoch=[]
best_training_loss = float("inf")
epoches = 50

for epoch in range(epoches):
    model.train()
    since = time.time()
    epoch_train_loss=[]

    # Fetch images and labels.
    for iteration, sample in enumerate(training_data_loader):

        optimizer.zero_grad()

        # The shape of img and mask is [Batch_size, 96, 96]
        img, mask = sample

        img, mask = img.to(device), mask.to(device)

        # Adding a dimension to img:[batch_size,1,96,96], ready to feed into 
        img = img.unsqueeze(1)

        # Calculate the expected mask
        # y_true = convert_to_4_channel(mask)

        # Write your FORWARD below
        y_pred = model(img)

        # Then write your BACKWARD & OPTIMIZE below
        loss = Loss(y_pred.float(), mask.long())
        train_loss.append(loss.item())
        epoch_train_loss.append(loss.item())

        loss.backward()
        optimizer.step()

    print('Epoch:{}/{} :: Ave_train_loss:{:.4f}'.format(epoch+1,epoches,np.me
    print('Epoch training time:{:.4f}'.format(time.time()-since))
    print('-'*10)
    plt.plot(train_loss)
    loss_training_epoch.append(np.mean(epoch_train_loss))

    # At the end of the epoch, do a test on the validation set and save the b
    if(loss_training_epoch[-1] < best_training_loss):
        best_training_loss = loss_training_epoch[-1]
        best_model = model
        torch.save(best_model.state_dict(), 'best_model.pth')
        print('save the best model!')

torch.save(model.state_dict(), "last_model.pth")
print(len(loss_training_epoch))
```

Testing

In [ ]:
```
import numpy as np
from torch.autograd import Variable
from matplotlib import pyplot as plt
```

In [ ]:
```
# In this block you are expected to write code to load saved model and deploy
# produce segmentation masks in png images valued 0,1,2,3, which will be used
data_path = './data/test'
num_workers = 0
batch_size = 2
i= 121

test_set = TestDataset(data_path)
test_data_loader = DataLoader(dataset=test_set, num_workers=num_workers,batch
model.load_state_dict(torch.load("best_model.pth"))
for iteration, sample in enumerate(test_data_loader):
    img = sample
    img = img.to(device)
    img = img.unsqueeze(1)
    img = model.forward(img)
    img = torch.argmax(img.squeeze(), dim=1)
    #visualise all images in test set
    img = img.data.cpu().numpy()
    plt.imshow(img[0,...].squeeze(), cmap='gray')
    plt.pause(1)
    cv2.imwrite('./data/test/mask/cmr{}_mask.png'.format(i), img[0])
    i+=1
    cv2.imwrite('./data/test/mask/cmr{}_mask.png'.format(i), img[1])
    i+=1
```

Evaluation

3.1 Dice Score

In [ ]:
```
import numpy as np
from matplotlib import pyplot as plt
from torch.utils.data import DataLoader

def categorical_dice(mask1, mask2, label_class=1):
    """
    Dice score of a specified class between two volumes of label masks.
    (classes are encoded but by label class number not one-hot )
    Note: stacks of 2D slices are considered volumes.

    Args:
        mask1: N label masks, numpy array shaped (H, W, N)
        mask2: N label masks, numpy array shaped (H, W, N)
        label_class: the class over which to calculate dice scores

    Returns:
        volume_dice
    """
    mask1_pos = (mask1 == label_class).astype(np.float32)
    mask2_pos = (mask2 == label_class).astype(np.float32)
    dice = 2 * np.sum(mask1_pos * mask2_pos) / (np.sum(mask1_pos) + np.sum(ma
    return dice
```

```
########################################### Loading & Evaluation validation da

path = 'best_model.pth'
val_data_path = './data/val'
num_workers = 0
batch_size = 4
val_set = TrainDataset(val_data_path)
val_data_loader = DataLoader(dataset=val_set, num_workers=num_workers, batch_

model.load_state_dict(torch.load(path))
model.eval()
for iteration, sample in enumerate(val_data_loader):
    val_img, val_mask = sample
    val_img, val_mask = val_img.to(device),val_mask.to(device)
    val_img = val_img.unsqueeze(1)
    val_img = model(val_img)
    val_pred = torch.argmax(val_img.squeeze(1), dim=1)
    val_mask, val_pred = val_mask.data.cpu().numpy(), val_pred.data.cpu().num

    ave_accuracy = []
    label_class = 1
    ave_accuracy.append(categorical_dice(np.array(val_pred), np.array(val_mas

    print('ave_accuracy:'+str(np.mean(ave_accuracy)))

    #visualise all images in test set
    plt.imshow(val_pred[0,...].squeeze(), cmap='gray')
    plt.pause(1)
```