FDL Kaggle Competition

Zhenning Li, Qiyun WU, Demouge Axel

January 2022

1 Introduction

In this Kaggle competition, we are solving the remote sensing problem using deep learning techniques. The main objective of this challenge is to segment images acquired by a small UAV (sUAV) at the area on Houston, Texas etc. The task is to design and implement a deep learning model in order to perform the automatic segmentation of such images. For this, our team implemented one commonly known deep learning model to predict and process labels for the various training dataset.

2 Solution steps

1) Download the training and test data set, use PIL, cv2 and PyTorch packages to transform, augment them etc. into tensor type;

2) Build training model based on current mature structures like UNet and ResNet, define every layer and step, every input and output dimension;

3) Splitting the data set into training and validation set, for choosing the best parameters (epochs, learning rate etc.) for countering overfitting problem. Also, transfer them into DataLoader type for further convenience;

4) Before training, resizing every graph we have to squared images in 256*256 or 512*512 pixel size for cutting down the GPU usage;

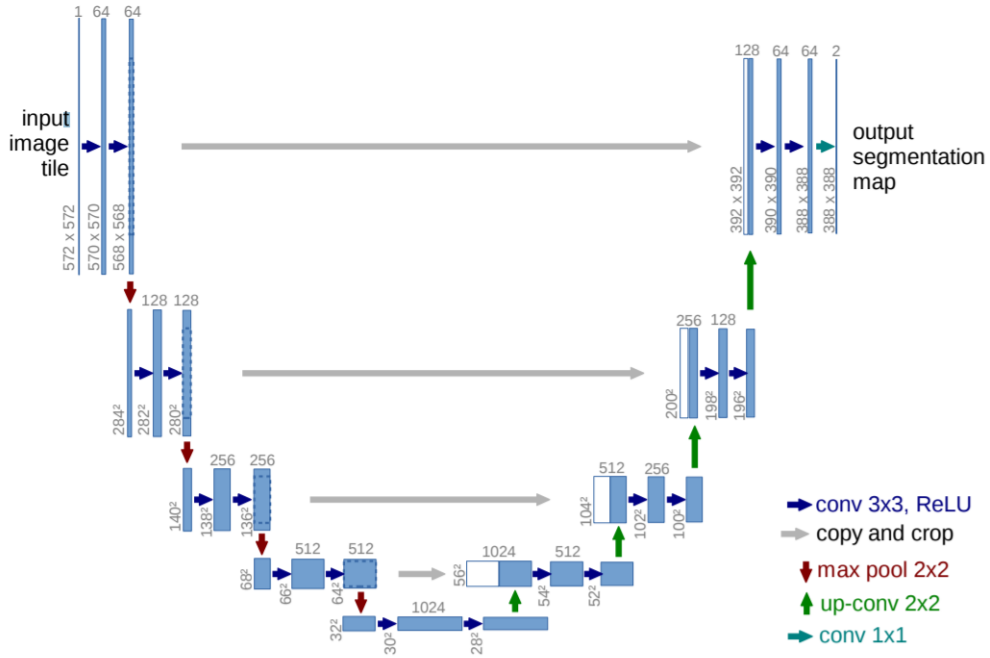
5) Begin training with the resized images, and store the model and loss function values for future use;

6) Plot the graph of loss function values for both training and validation set, to determine the best epochs;

7) Make predictions on test images and store them, and run the submission.py to generate the final submission.csv file;

8) Make submission and see the result, if not good, go back to 2) to reselect model parameters.

3 Solution with U-Net method



3.1 Brief introduction to U-Net

The above picture is how U-Net works. This architecture consists of three sections: The contraction, The bottleneck, and the expansion section. The contraction section is made of many contraction blocks. Each block takes an input applies two 3*3 convolution layers followed by a 2*2 max pooling. The number of kernels or feature maps after each block doubles so that architecture can learn the complex structures effectively. The bottom-most layer mediates between the contraction layer and the expansion layer. It uses two 3*3 CNN layers followed by 2*2 up convolution layer. Next, the expansion section also consists of several expansion blocks, and each block passes the input to two 3*3 CNN layers followed by a 2*2 up-sampling layer. Also after each block number of feature maps used by convolutional layer get half to maintain symmetry. However, every time the input is also get appended by feature maps of the corresponding contraction layer. This action would ensure that the features that are learned while contracting the image will be used to reconstruct it, and the number of expansion blocks is as same as the number of contraction block. After that, the resultant mapping passes through another 3*3 CNN layer with the number of feature maps equal to the number of segments desired.

3.2 Our structure for U-Net

Similar to the structure above, we used PyTorch to set up the model step by step:

1) Set up the DoubleConv class, which is used to to the horizontal transformation in graph (where blue arrows point);

```
class DoubleConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.double_conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
```

```
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )
    def forward(self, x):
        return self.double_conv(x)
```

---

2) Set up the Down class, which is what the red arrow points and it is used to extract the features hidden inside the data;

```
class Down(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.maxpool_conv = nn.Sequential(
            nn.MaxPool2d(2),
            DoubleConv(in_channels, out_channels)
        )
    def forward(self, x):
        return self.maxpool_conv(x)
```

---

3) Set up the Up class, which is what the green arrow points and it is used to do up convolution;

```
class Up(nn.Module):

    def __init__(self, in_channels, out_channels, bilinear=True):
        super().__init__()
        if bilinear:
            self.up = nn.Upsample(scale_factor=2, mode='bilinear',
                                                  align_corners=True)
        else:
            self.up = nn.ConvTranspose2d(in_channels, in_channels // 2,
                                                  kernel_size=2, stride=2)
        self.conv = DoubleConv(in_channels, out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)
        diffY = torch.tensor([x2.size()[2] - x1.size()[2]])
        diffX = torch.tensor([x2.size()[3] - x1.size()[3]])
        x1 = F.pad(x1, [diffX // 2, diffX - diffX // 2,
                        diffY // 2, diffY - diffY // 2])
        x = torch.cat([x2, x1], dim=1)
        return self.conv(x)
```

---

4) Set up the output layer, we can input the number of out channels we want here;

```
class OutConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(OutConv, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=1)

    def forward(self, x):
        return self.conv(x)
```

---

5) Now we can start to build the model;

```
class UNet(nn.Module):
    def __init__(self, n_channels, n_classes, bilinear=False):
        super(UNet, self).__init__()
        self.n_channels = n_channels
        self.n_classes = n_classes
        self.bilinear = bilinear
        self.inc = DoubleConv(n_channels, 64)
        self.down1 = Down(64, 128)
        self.down2 = Down(128, 256)
        self.down3 = Down(256, 512)
        self.down4 = Down(512, 1024)
        self.up1 = Up(1024, 512, bilinear)
        self.up2 = Up(512, 256, bilinear)
        self.up3 = Up(256, 128, bilinear)
        self.up4 = Up(128, 64, bilinear)
        self.outc = OutConv(64, n_classes)

    def forward(self, x):
        x1 = self.inc(x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)
        x = self.up1(x5, x4)
        x = self.up2(x, x3)
        x = self.up3(x, x2)
        x = self.up4(x, x1)
        logits = self.outc(x)
        return logits
```

---

6) Then we define the DataLoader and now we can train the model.

```
class ISBI_Loader(Dataset):
    def __init__(self, train_data_path, train_mask_path):
        self.train_data_path = train_data_path
        self.train_mask_path = train_mask_path
        self.imgs_path = glob.glob(os.path.join(train_data_path, '*.jpg'))
        self.mask_path = glob.glob(os.path.join(train_mask_path, '*.png'))

    def augment(self, image, flipCode):
```

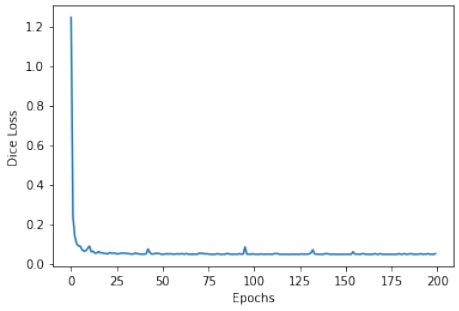
```
        flip = cv2.flip(image, flipCode)
        return flip

    def __getitem__(self, index):
        data_path = self.imgs_path[index]
        mask_path = self.mask_path[index]
        image = cv2.imread(data_path)
        label = cv2.imread(mask_path, 0)

        if label.max() > 1:
            label = label / 255
        if image.max() > 1:
            image = image / 255

        flipCode = random.choice([-1, 0, 1, 2])
        if flipCode != 2:
            image = self.augment(image, flipCode)
            label = self.augment(label, flipCode)

        image = torch.tensor(image)
        image = image.permute(2, 0, 1)
        label = torch.tensor(label)
        return image.numpy(), label.numpy()

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

---

4 Outcome and submissions

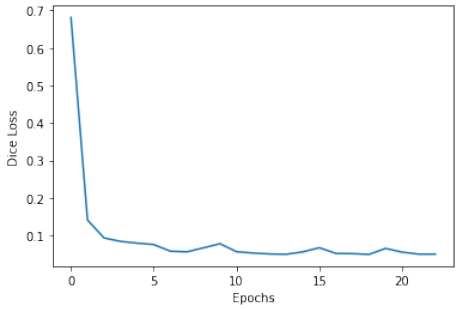
We have tried many times to train the model and draw the loss function value graph. The followings are our results:

1) Resized 256*256, epochs=200, lr=0.00001, resized after prediction:



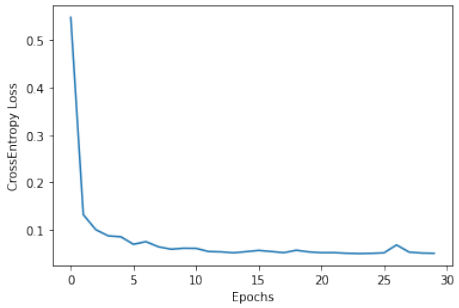
Submission score: 0.6033

2) Resized 256*256, epochs=25, lr=0.00001, no normalization (255), resized after prediction:



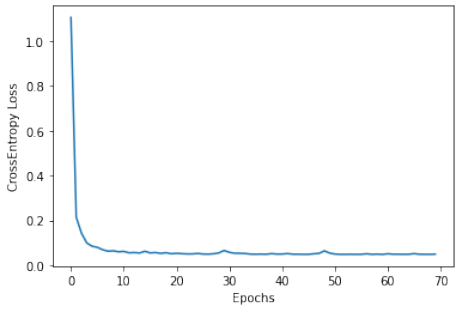
Submission score: 0.6201

3) Resized 512*512, epochs=30, lr=0.0001, no normalization (255), not resized after prediction:



Submission score: 0.6518

4) Resized 512*512, epochs=60, lr=0.00001, no normalization (255), not resized after prediction:



Submission score: 0.6677

5 Some results & thoughts

1) After trying for about 20 models, we found that, the best model has the following parameters:

epochs=50,

learning rate=0.00001

padding=1

train image size=512*512

no normalization

no resizing in predicted masks

shuffle=True in DataLoader

Do image augmentation in DataLoader by randomly selecting images to revert to generate more training images

2) When training with image size 1024*1024, it's rather easy to get very overfitted, and we think this problem is more serious here because we have 24 classes. If there are only 2-3 classes, using a high resolution image to train would be far better, and U-Net model was initially used to segment medical images which indeed have less classes.

6 Model problems and thoughts in U-Net

1) To counter the overfitting problem, we split the dataset into training and validation parts, and we choose the number of epochs to be the x-value of the best intersection point. In theory this is the best method upon the given learning rate. The problem is that, if the model does not overfit, it seems to be hard for it to learn the details of the training dataset, so the submission file have a rather low score; if the model slightly overfits, then although the submissino score is higher, it only can classify one or two kinds (class 0 and class 1 only!).

2) While doing the model, according to the original paper, there was no padding (padding=0), and I think this is because padding was not popular at that time. We tried to train the model with padding=0 and padding=1, with kernel_size=3 we found that the model performs better when padding=1 (so that the image size does not change in each horizontal or down layer).

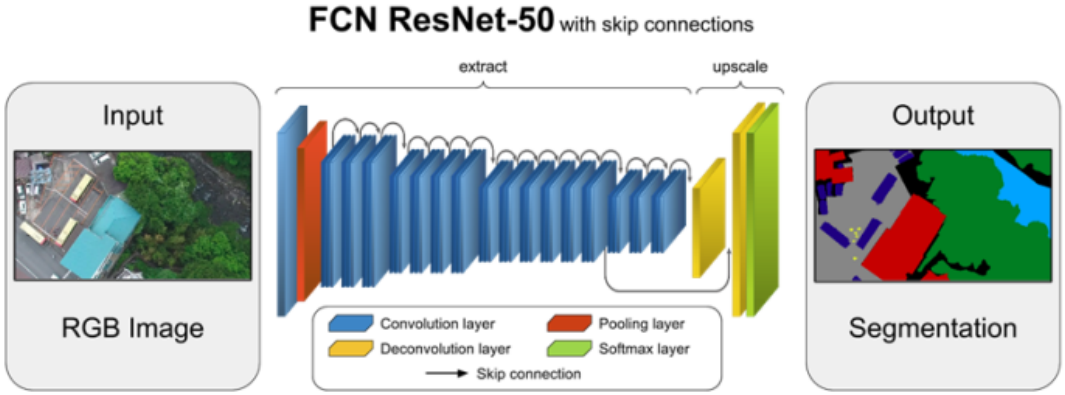
3) To increase the training speed, we resized the image to 512*512 pixels, and indeed the training session was much faster than directly train the 4000*3000*3 pixel image. But I think maybe we can make the machine learn better with the original images after image augmentation (with Google Colab Pro GPU of course!), and handling huge images of any size is exactly one of the advantages of U-Net model.

4) In downsizing the images, we reshaped every image to be squared, and we used two ways to do it: filling the difference between height and width with 0 or cropping the image based on the smaller value between height and width and then downsize linearly. We found that the submission score for the first kind is a bit higher. It accords with the theory, since the first kind retains more comprehensive information for the machine to learn. It's also easier for the second cropping method to have a lower possibility to classify the image parts to be property roof since most of the roofs are in the upper part of the image.

5) Although unnecessary, before reading the submission.py file, we still did some upsizing work on the predicted test masks. We used two ways: Nearest and Linear interpolation. We found that, using linear interpolation method can have a slightly better submission score. This maybe because if we use nearest, the training mask will have more pixels to be 1 (the second most possible class), but with linear interpolation, many pixels will just become 0 (the most possible class).

7 Further Discussions

7.1 Other Solution with FCN Resnet 50



In order to improve our results, other networks architectures already existing could have been used. In paticular, this is the case of the FCN Resnet 50 network which is a fully convolutionnal network. This network only uses convolutionnal and deconvolution layers. It uses a softmax function to learn the different classes probabilities. This model uses semantic segmentation modelling. This means that it gives us a pixelwise mask of the different classes present on the image. This is to be opposed with bounding boxes like networks which seek for regions of interest on the image and bound them in boxes.

7.2 Other results using different loss functions

For this exercise, the loss function used was the mean dice function. This metric is well suited to measure the loss for image segmentation problem. However, other loss functions exist and we might have used them in order to tackle different problems or simply improve our result :

1) The pixel accuracy could have been used to measure the accuracy of our model.

$$L = \frac{\sum_{i=0}^{K} p_{ii}}{\sum_{i=0}^{K} \sum_{j=0}^{K} p_{ij}}$$

With K the number of classes and p_{ij} the amount of pixels of class i predicted as class j.

2) Mean intersection over union which represents the portion of pixels that are intersection of the predicted segmentation of map and the ground truth map, with the union of predicted and ground truth segmentation maps, averaged on all the classes. Compared to the dice loss, this loss gives less weight to true positives and rewards as much those as the true negatives.

3) Tversky Loss : this loss is very similar to the dice loss as well but allow us to add a custom weight for the false positives and the false negatives (named hereafter β).

$$L = 1 - \frac{1 + TP}{1 + TN + \beta * FP + (1 - \beta) * FN}$$

8 Some finding in U-Net

When processing the data, it makes no difference to do normalization or no (divided by 255). We think this is because in PyTorch when you transfer the image into tensor type, it is automatically normalized.

9 Some difficulties & thoughts in U-Net

1) One of the most frequently seen error message is about tensor dimension error while doing the matrix multiplication. This is a major problem in our project because it's hard for novices like us to determine every tensor size in every step. For example, when you use DataLoader to train the model, with the batch size=5, then the training data set has the dimension of [5, 3, 255, 255], and now you can use the mask as the label (which has the size of [5, 1, 255, 255] or [5, 255, 255], as the model supports zero dimension in color channel) to train. In the challenge, we used many times the function tensor.squeeze() and tensor.unsqueeze() to reshape the tensor to satisfy the parameter needs.

2) Another problem is related to CUDA. Since we are using Google Colab Pro to do the challenge, when we trained the model sometimes there would be an error message saying that we are using paralleled CUDA which will result in a conflict. After spending some hours into it I finally found a solution: add .to('cuda') every time when processing any models or data or labels.

3) Every time when passing the tensor parameter to any functions, remember to change the type of the tensor to their necessary type. For example, since CrossEntropyLoss only supports LongTensor, you cannot define it to be torch.float32 or don't define it (because torch.float32 is default type), so every time when processing the tensor, seems that you have to change the type by doing tensor.long().

4) When processing the training data (train and test images), it's better to only use one package, either only torchvision.datasets, or cv2, or PIL.Image. Make sure don't use them together, and there is no need to use another one just for one of its easy-to-realize function. This is because, PyTorch and PIL will automatically normalize the image while cv2 will not, so we have to add: if label.max()>1: label = label/255.

10 Summary

In this Kaggle competition, we learned how to deal with image segmentation problems with PyTorch structure from model building, dataset processing, DataLoader construction, training, loss function value plotting, to the final parameter determination. We mainly used U-Net method to train and predict, and the result does not have a top ranking, but around average. We think this is mainly because of the parameter tuning problem, which is what we should enhance in later projects.

It is also important to note that on almost all of our models we got scores between 0.63-0.68 when submitting to Kaggle. This was mainly due to the fact that most of the information in the training set could not be learned by the machine if the parameters were not set well.

Grateful to have had this opportunity. Please have a look at the notebook for more information.