

Dataset Familiarization

As mentioned in the description, we have a dataset consisting of 1000 sampled polyp images along with their corresponding ground truth masks. It's important to note a detail highlighted in the data description: the square region captured by the medical camera used by the doctor appears blackened in the image to avoid confusing our model.

Another crucial aspect is that both the images and their masks must share the same dimensions. Our dataset exhibits varying sizes, ranging from 332x487 to 1920x1072 pixels. Due to computational constraints, we opted for a conservative approach in resizing, avoiding larger dimensions. Additionally, most of the models we employ necessitate image dimensions divisible by 32. Consequently, we selected the dimensions of our images to be 320x320.

Preprocessing

One approach is to convert our images to grayscale. However, considering that most deep learning models accept RGB images as inputs, there is no need for grayscale conversion. Gray scaling the pictures would result in information loss, which is undesirable.

It's important to note that we have divided our data into three parts: a training set (800 samples), a validation set (100), and a test set (100). Instead of directly splitting the images, we have introduced a dataframe with 1000 rows, where each row contains two columns: one for the image path and the other for the mask path. We then split this dataframe.

Since we aim to use deep learning models, we employ dataloaders. Dataloaders take a class of data and a batch number, providing the data batch by batch to the model.

To use dataloaders effectively, we need to convert our images to a customized data class. These data classes should include `__init__`, `__len__`, and `__getitem__` methods. The arguments for these methods are our dataframe and a transform function to be applied to each image while being used by the model. The transformation we wish to apply to our data involves resizing the images and augmenting them by flipping horizontally or vertically with a probability of 0.5. In our `__getitem__` method, for the desired index in our dataframe, we retrieve the image and its corresponding mask from the specified paths, transform them, and return them in a dictionary.

```

▶ device = torch.device("cuda" if torch.cuda.is_available else "cpu")

# Split the data set into three parts : train, validation, test

train_df, val_test_df = train_test_split(df, test_size=0.2, random_state = 100)
val_df, test_df = train_test_split(val_test_df, test_size=0.5, random_state = 100)

# Change df to dataset
train_dataset = MyDataset(train_df, train_transforms)
val_dataset = MyDataset(val_df, test_transforms)
test_dataset = MyDataset(test_df, test_transforms)

# Create dataloader with 32-sized batches
BATCH_SIZE = 32
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE)
print(f'len(train_loader): {len(train_loader)}')

```

After defining each of the three datasets as instances of this new customized data class, we can define a dataloader for each dataset, specifying the desired number of batches (chosen to be 32 in our case). With our dataloaders in place, we are now ready to train our models.

Model Selection and Training

We employed three distinct models in this project, utilizing the `segmentation_models_pytorch` library—a well-known and suitable tool for segmentation tasks. We selected three different models from this library, each of which will be discussed. Given the limited size of our dataset, we opted for pretrained models, and, in all cases, we utilized ImageNet weights for our models.

U-net with ResNet50 Encoder

The U-Net architecture, originally designed for biomedical images, serves as the basis for two of our chosen models. In the first instance, we employed ResNet50 as the encoder. This choice is influenced by the historical use of U-Nets in biomedical applications. Additionally, since our images are in RGB format, we adopted a model configured with 3 input channels to appropriately handle the color information.

```

▶ # Create the model with Unet(resnet50)

class_size = 1
model = smp.Unet(
    encoder_name="resnet50",
    encoder_weights="imagenet",      # use `imagenet` pre-trained weights for encoder initialization
    in_channels=3,                  # Cause it's RGB
    classes=class_size,             # our mask is binary
)

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

```

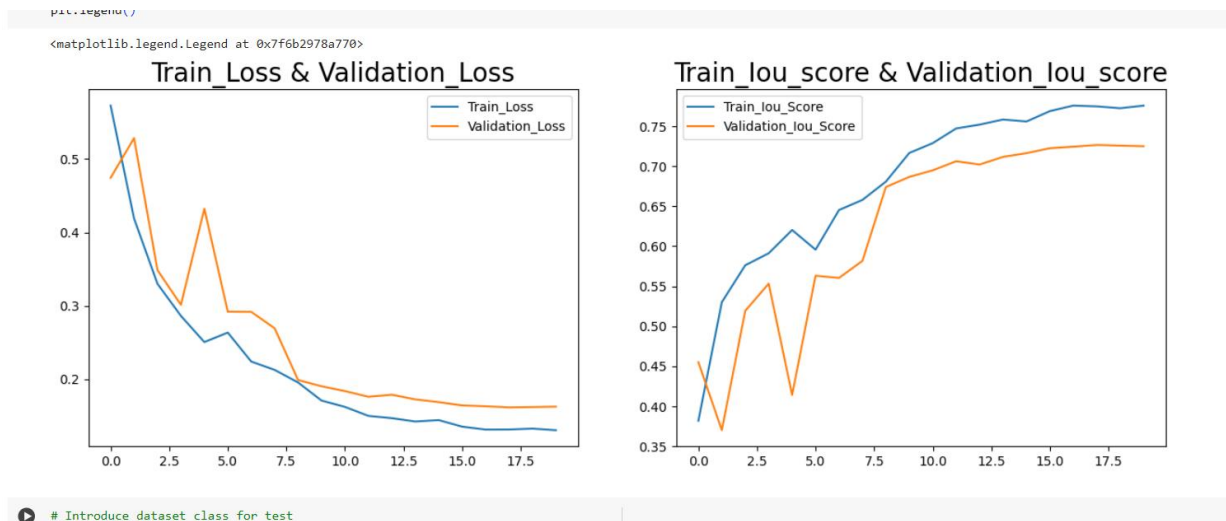
The training function iterates over the training dataloader, conducting forward and backward passes through the model. The computed loss is then backpropagated, and model parameters are updated using the Adam optimizer. Our chosen loss function is the Dice Loss. Moreover, the Intersection over Union (IoU) score is calculated to assess segmentation accuracy. The learning rate is adjusted using the

step learning rate scheduler. The training loop is structured to monitor and return the average loss and IoU score for each epoch. These values are crucial for visualizing whether our training converges over different epochs.

Logs are maintained for both training and validation losses, along with IoU scores, across epochs. This comprehensive tracking system allows us to monitor the model's progress effectively. The model is saved after each epoch, and in the event of improved validation loss, the best model is also saved separately. This practice ensures the retention of the best-performing model during training.

To prevent overfitting, early stopping is implemented with a patience parameter set to 5. If the validation loss does not improve over a consecutive number of epochs (in this case, 5 epochs), training is halted early. This approach minimizes unnecessary computation and guarantees that the model does not overfit to the training data.

As mentioned, we calculated logs for different epochs which results in following visualization:



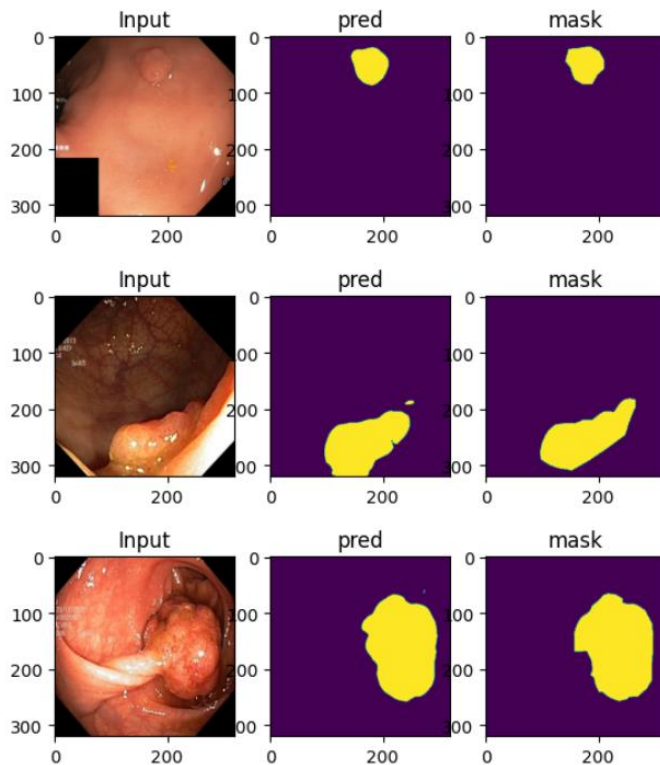
In the initial epochs, both our loss and IoU scores exhibit instability and rapid fluctuations in both the training and validation datasets. However, as the epochs progress, the loss consistently decreases, and the validation set and test set performances converge towards each other. This trend indicates that overfitting is not occurring. The IoU scores also follow this pattern, affirming the stability and generalization capability of our model.

After completing the training phase, the next step is evaluation. To accomplish this, we utilize our test data loader, making predictions for each batch. Subsequently, we compare these predictions with the actual 'y,' which represents the masks of the images in the respective batch. Various metrics, including IoU, accuracy, recall, precision, and Dice (F1-score), are computed for each batch. After computing these metrics for every batch, we calculate their averages, reporting them as the metrics for the entire test set.

Another aspect to consider is the threshold parameter. We compute these metrics for the test data using different thresholds, with the results summarized as follows:

threshold: 0.3	Dice: 0.894	IoU Score: 0.809	precision: 0.904	Recall: 0.887	Acc: 0.971
threshold: 0.4	Dice: 0.894	IoU Score: 0.810	precision: 0.906	Recall: 0.886	Acc: 0.971
threshold: 0.5	Dice: 0.894	IoU Score: 0.810	precision: 0.907	Recall: 0.885	Acc: 0.971
threshold: 0.6	Dice: 0.894	IoU Score: 0.810	precision: 0.909	Recall: 0.883	Acc: 0.971
threshold: 0.7	Dice: 0.894	IoU Score: 0.810	precision: 0.911	Recall: 0.882	Acc: 0.971

As observed, the threshold did not significantly impact our evaluation results. The results will be discussed further in upcoming parts. But we also visually compared our prediction with the result for some of the data.



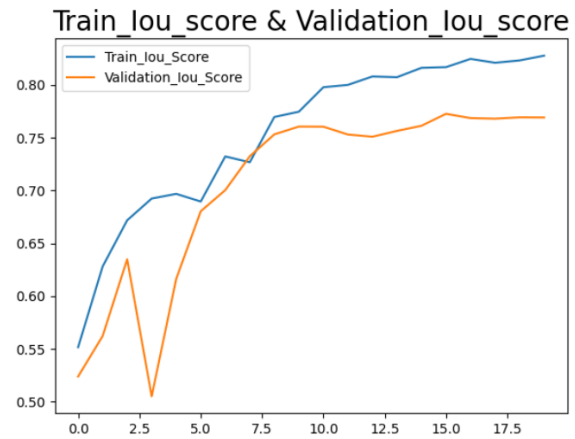
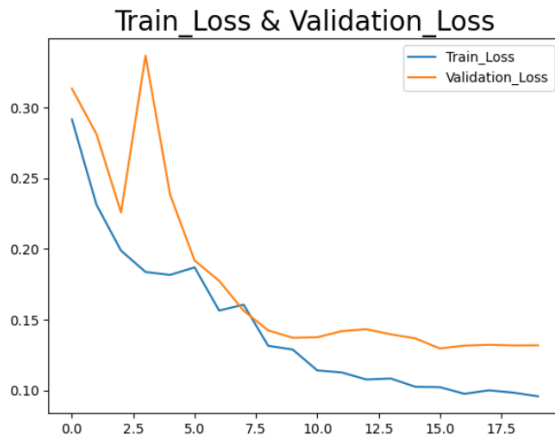
Feature Pyramid Network (FPN) with ResNet18

Another popular deep learning model for segmentation is FPN. We employed this model with ResNet18, utilizing the same pretrained weights from ImageNet as mentioned earlier. The input channels and classes remain consistent with the previous configuration.

```
# FPN pretrained model
model2 = smp.FPN(
    encoder_name="resnet18", # You can choose other encoders like resnet34, resnet50, etc.
    encoder_weights="imagenet", # Use pre-trained weights on ImageNet
    in_channels=3, # Number of input channels (e.g., for RGB images)
    classes=1, # Number of output channels/classes for segmentation
)
```

The training and validating process is similar to the previous model. We just show the plots. The training and validation processes are similar to those of the previous model. We will present only the plots.

```
> <matplotlib.legend.Legend at 0x7c84f9a47a60>
```



As previously, the plots indicate the absence of overfitting, and the model appears stable. However, in comparison to the previous model, there is a more noticeable divergence between the validation and training lines towards the end. This discrepancy may suggest a potential risk of overfitting with further epochs.

The metrics for this model are as follows:


```
ACC: {sample[ acc ]:.5T} )
```

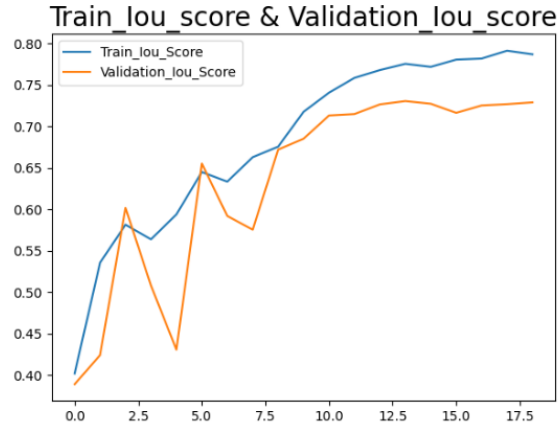
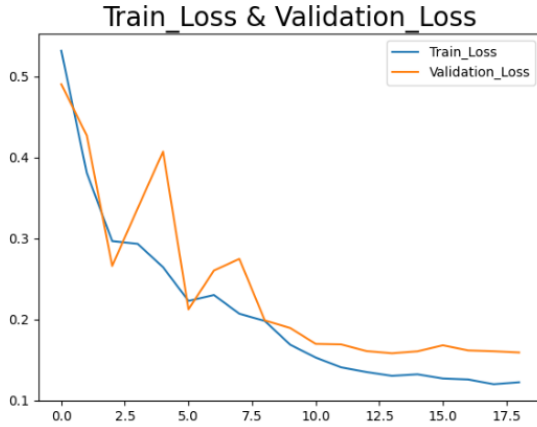
threshold: 0.3	Dice: 0.901	IoU Score: 0.821	precision: 0.917	Recall: 0.889	Acc: 0.973
threshold: 0.4	Dice: 0.901	IoU Score: 0.821	precision: 0.920	Recall: 0.886	Acc: 0.973
threshold: 0.5	Dice: 0.901	IoU Score: 0.821	precision: 0.923	Recall: 0.883	Acc: 0.973
threshold: 0.6	Dice: 0.901	IoU Score: 0.820	precision: 0.925	Recall: 0.881	Acc: 0.973
threshold: 0.7	Dice: 0.900	IoU Score: 0.820	precision: 0.928	Recall: 0.878	Acc: 0.973

Which is slightly better than the previous model.

U-net with ResNet101 Encoder

Our final model is once again a U-Net, with ResNet101 serving as its encoder. ResNet101 is deeper and has more parameters compared to ResNet50. The remaining aspects remain consistent with the previous models, and the training and validation procedures follow the same protocol. We present the plot and metrics for evaluation.

 <matplotlib.legend.Legend at 0x7c84f98619f0>



This plot, like the ones from the previous models, illustrates the absence of overfitting, indicating the stability of our model. However, in this plot, it appears that with additional epochs, the convergence towards each other and improved loss and IoU scores could have been achieved. This is logical considering that ResNet101, with its increased number of parameters, may require more epochs to converge effectively. Unfortunately, due to our computational constraints, we were unable to extend the learning further. The metrics are detailed as follows:

```
threshold: 0.3    Dice: 0.898    IoU Score: 0.817    precision: 0.909    Recall: 0.891    Acc: 0.972
threshold: 0.4    Dice: 0.899    IoU Score: 0.817    precision: 0.912    Recall: 0.889    Acc: 0.972
threshold: 0.5    Dice: 0.899    IoU Score: 0.818    precision: 0.915    Recall: 0.887    Acc: 0.972
threshold: 0.6    Dice: 0.899    IoU Score: 0.818    precision: 0.917    Recall: 0.885    Acc: 0.972
threshold: 0.7    Dice: 0.899    IoU Score: 0.818    precision: 0.920    Recall: 0.883    Acc: 0.972
```

The Average of Models

Another approach involves utilizing the predictions from all three models, averaging them, and using this combined value as the predicted result. Various metrics are then measured for this averaged approach, employing different thresholds. The results are presented below:

```
Dice: 0.893    IoU Score: 0.807    precision: 0.870    Recall: 0.922    Acc: 0.970
Dice: 0.897    IoU Score: 0.815    precision: 0.908    Recall: 0.890    Acc: 0.972
Dice: 0.898    IoU Score: 0.815    precision: 0.913    Recall: 0.886    Acc: 0.972
Dice: 0.898    IoU Score: 0.815    precision: 0.917    Recall: 0.882    Acc: 0.972
Dice: 0.893    IoU Score: 0.809    precision: 0.946    Recall: 0.849    Acc: 0.972
```

Comparing Results

The best results achieved by each model are presented in the table below:

	Dice	IoU	Precision	Recall	Accuracy
U-Net(ResNet50)	0.894	0.810	0.906	0.886	0.971
FPN(ResNet18)	0.901	0.821	0.917	0.889	0.973
U-Net(ResNet101)	0.899	0.818	0.917	0.885	0.972
Averaged Model	0.898	0.815	0.913	0.886	0.972

As we are aware, accuracy and precision may not be suitable metrics for segmentation tasks due to the imbalanced nature of datasets, especially when a significant portion of the image represents the background. Therefore, metrics such as Dice and IoU are preferred, as they provide more meaningful evaluations for segmentation performance.

In our models, the best performance is observed for FPN, which could be attributed to two factors. Firstly, the FPN model may be better suited for our data compared to U-Net. Secondly, the shallower ResNet with fewer parameters might be more suitable for our data, as it allows for better fine-tuning.

When comparing our results with those presented in the paper, we have achieved commendable performance. Our single encoder-decoder models outperform all single encoder-decoder models from the paper. However, the double encoder-decoder models in the paper outperform the ones in our project.

Conclusion

Our segmentation models, including U-Net with ResNet50, FPN with ResNet18, and U-Net with ResNet101, exhibited robust performance on our polyp dataset. Notably, the FPN model showed the best results, outperforming even the double encoder-decoder models from a comparative paper. This achievement is noteworthy as our models are computationally lighter, emphasizing the efficiency of our approach in obtaining superior results.