Explanation of My approaches with results

Import All Required Library And Dependency :

First, Import some Basic libraries Like – os, glob,time,numpy, pandas,cv2,imageio, torch, matplotlib,tqdm,sklearn,albumentation

And the model – segmentation_model_pytorch

Set up Colab GPU Runtime Environment :

```
!pip install segmentation-models-pytorch
!pip install -U git+https://github.com/albumentations-team/albumentations
!pip install --upgrade opencv-contrib-python
```

Setup Configurations :

This is the remote for our model , where I tune the basic hyperparameters

Load the data :

In this step I create a function to load the given data using glob library. And loads the 20 .

Apply Augmentation On data :

Here Apply Augmentation like Horizontal Flip, Vertical Flip, rotation on the given data to increase the no of data . and now we have 80 data.

Create DataFrame to store Path for images and masks :

After applying Augmentation save the segmented image in different directory and save all the path of images and Masks to read them in furture.

Show Some Given Data :

To take a overview of the given data we show some of them.

Create Custom Dataset :

Rason – Get image and mask pair according to the E1 index.

Create Show_Image Function :

Create a Function to show data(image and mask pair in E1 index) and output_masks(if required)

Load Dataset Into Batches :

To train the model, load the total dataset into batches.

Create Segmentation Model :

Create the model using pre-build model segmentation_models_pytorch , here we use the hyperparameter as

Encoder_name = ,encoder_weights=,in_chsnnels=3,activation=None

And for losses use, DiceLoss and BCEWithLogitsLoss

Create Train and Validation Function :

To train the model, create a train function and give us training loss.

And create an evaluation function to evaluate our model in ach epochs.

Train The Model :

Now it is time to train our model, and we use Adam optimiser.

Plot Training And Validation Losses :

After training the model, we plot the training loss and validation loss for each epochs .And see that they decrease in each epoch.

Show image , Ground_truth and Model_output for all validation data:

To visualise our prediction, we show the input image, target image and our model output_image for all validation data.

Calculate Some metrics:

For Checking our model, we calculate some metrics as –

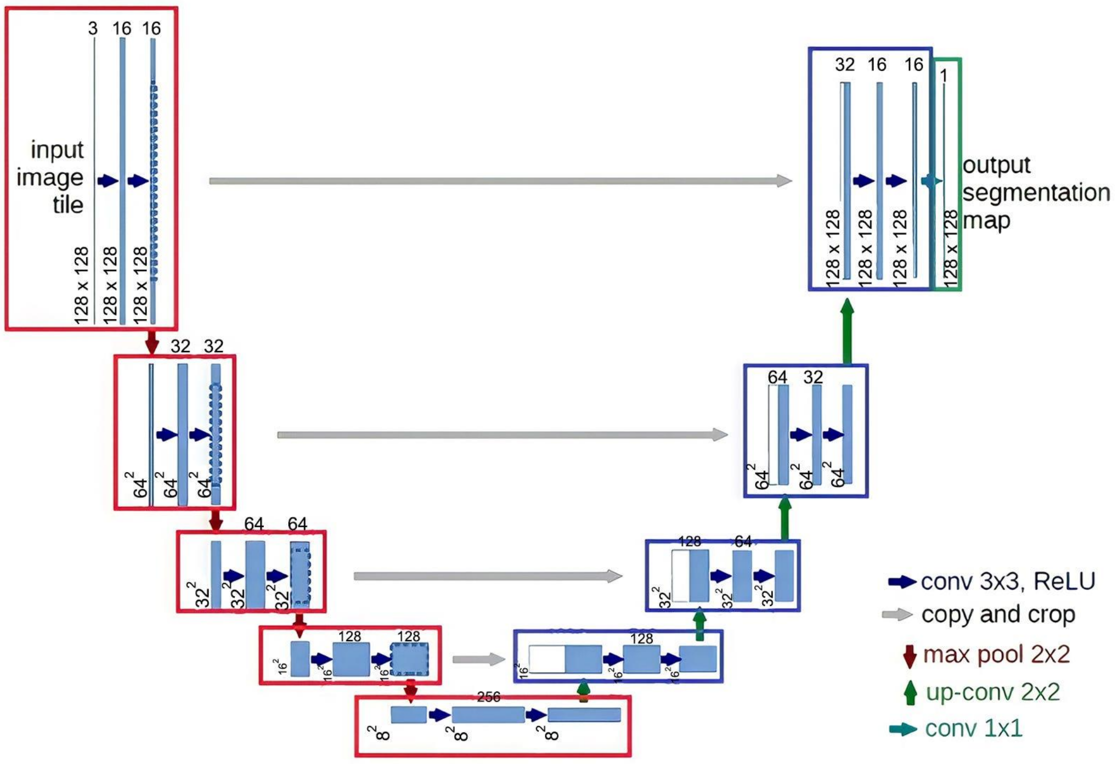
- Accuracy Score = 99.48 %
- IOU Score = 80.15%
- F1 Score = 88.96 %
- Prevision = 83.10%
- Recall = 95.75%

Model architecture and reason for selecting this architecture

Model:-

In recent times, whenever we wish to perform image segmentation, the first model we think of is the U-Net. It has been revolutionary in performance improvement compared to previous state-of-the-art methods. U-Net is an encoder-decoder convolutional neural network with extensive medical imaging, autonomous driving, and satellite imaging applications. However, it is essential to understand how the U-Net performs segmentation as all novel architectures post-U-Net are developed on the same intuition.

U-Net Architecture:-



Reason for selecting this architecture:-

1. Proven Success: U-Net has excelled in diverse image segmentation tasks since 2015.

2. Context and Localization: U-Net efficiently captures both global context and local details.

3. Skip Connections: U-Net utilises skip connections to preserve spatial information during training.

4. Fully Convolutional Network (FCN): U-Net operates directly on images of any size.

5. Efficiency: U-Net can efficiently segment images of size 512x512 with modern GPUs.

6. Adaptability: U-Net can be modified to suit specific requirements and datasets.

7. Robustness to Variability: U-Net demonstrates robustness to variations in image characteristics.

8. Semantic Segmentation Capability: U-Net assigns pixels to specific classes.

9. Scalability: U-Net's modular architecture makes it scalable to handle different input sizes.

10. Interpretability: U-Net's architecture is intuitive, facilitating a better understanding of predictions.

Hardware requirements for running the code

To Complete the task, I use GOOGLE Colab. The requirements are,

GPU Runtime: Ensure that your Google Colab notebook is set to use a GPU runtime. You can change this setting by going to "Runtime" -> "Change runtime type" -> select "GPU" from the dropdown menu. This is crucial for faster training of deep learning models.

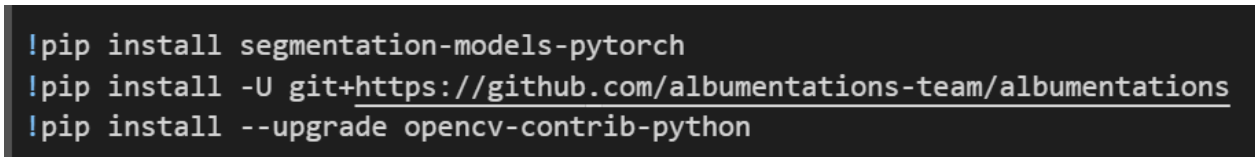
In this case, my Bad. I don't know why, in my system, GPU 'cuda' is not available, so CPU will be used, but the code can use GPU when the 'cuda' is available.

Hardware Requirements: Google Colab provides free access to a GPU, which is sufficient for many deep-learning tasks.

Internet Connection: Google Colab requires an active internet connection since it's a cloud-based platform. Ensure that you have a stable internet connection to work seamlessly.

Any other specific installations required

In the stating of code, we take some installation those are setup colab GPU Runtime Environment and the uses model,



```
!pip install segmentation-models-pytorch
!pip install -U git+https://github.com/albumentations-team/albumentations
!pip install --upgrade opencv-contrib-python
```

Image Segmentation:-

let us understand what image segmentation is. Computer Vision has been one of the many exciting applications of machine intelligence. It has numerous applications in today's world and makes our lives easier. Two of the most common computer vision tasks are image classification and object detection.

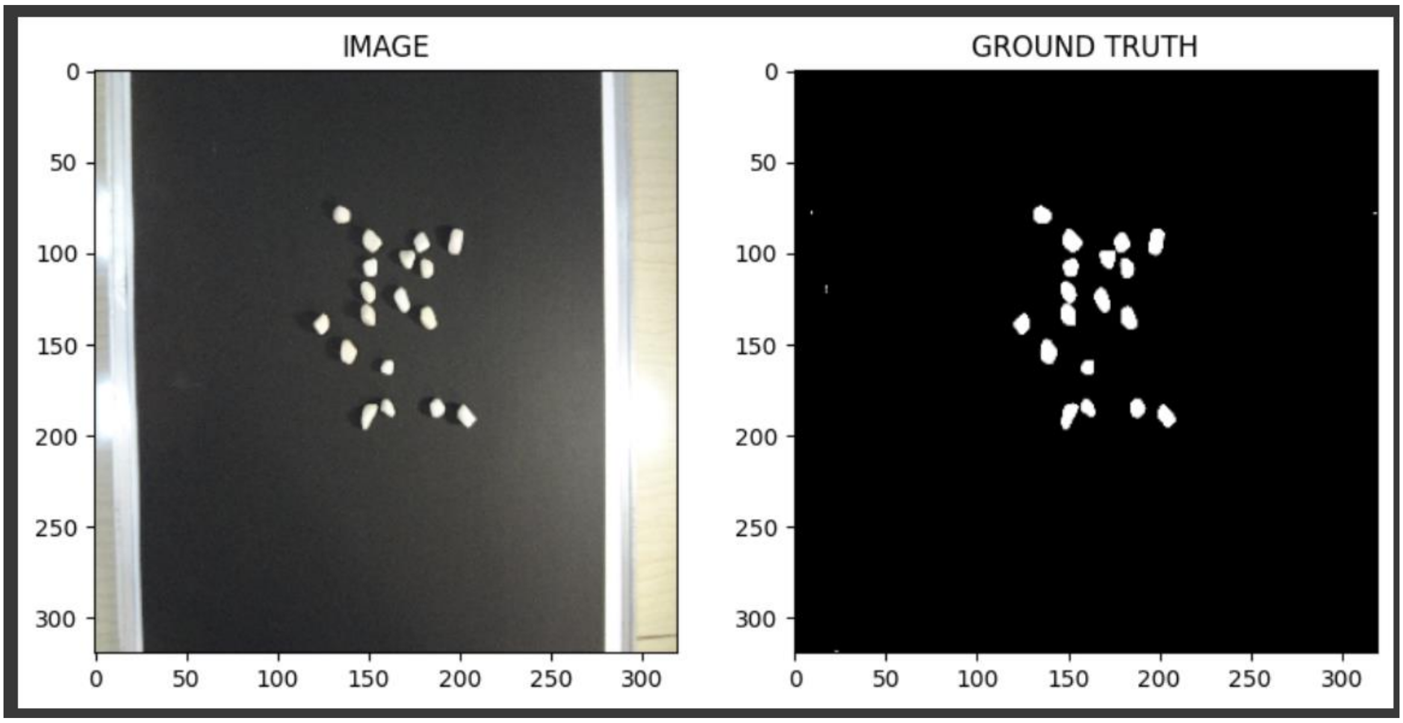
Image classification for two classes involves predicting whether the image belongs to class A or B. The predicted label is assigned to the entire image. Classification is helpful when we want to see the class in the image.

On the other hand, object detection takes this further by predicting the object's location in our input image. We localise objects within an image by drawing a bounding box around them. Detection is helpful in locating and tracking the contents of the image.

Image segmentation is the combination of classification and localisation.

Image segmentation involves partitioning the image into smaller parts called segments. Segmentation is used to understand what is given in an image at a pixel level. It provides fine-grained information about the image as well as the shapes and boundaries of the objects. The output of image segmentation is a mask where each element indicates which class that pixel belongs to.

In our Case,



Given above, on the left is our input image of Some stone. Our task is to separate the stone from the background. So we have two output classes – stone [1] and background [0]. However, to separate this stone from its background, we need to know its exact location in the image. We are to find answers to two questions-

1. "What" is in the input image?

Ans: Stone and Background

2. "Where" is that object in the input image?

Ans: The location of the Stone in the image

Image segmentation solves the above problem pixel by pixel. We wish to group similar pixels and separate dissimilar pixels. At each pixel, we will classify whether that pixel is part of the stone or the background. Thus, all the pixels that our model predicts belong to the stone will have the label 1, and the remaining pixels will have the label 0. In this process, we would have created a mask of our input image as shown above, and at the end of this pixel-wise classification, we also would have detected the Stone's exact location in our image.

Use Segmentation_model_pytorch :-

Use U-Net Model:-

Unit is a full convolution neural network for image semantic segmentation. Consists of *encoder* and *decoder* parts connected with *skip connections*. Encoder extracts features of different spatial resolutions (skip connections), which are used by the decoder to define an accurate segmentation mask. Concatenation is used to fuse decoder blocks with skip connections.

Parameters:

- **encoder_name** – Name of the classification model that will be used as an encoder (a.k.a backbone) to extract features of different spatial resolution
- **encoder_depth** – Several stages used in the encoder in the range [3, 5]. Each stage generates features two times smaller in spatial dimensions than the previous one (e.g. for depth 0, we will have features with shapes [(N, C, H, W),], for depth 1 - [(N, C, H, W), (N, C, H // 2, W // 2)] and so on). Default is 5
- **encoder_weights** – One of **None** (random initialisation), **"imagine"** (pre-training on ImageNet) and other pre-trained weights (see a table with available weights for each encoder_name)
- **decoder_channels** – List of integers which specify **in_channels** parameter for convolutions used in decoder. The length of the list should be the same as **encoder_depth**
- **decoder_use_batchnorm** – If **True**, BatchNorm2d layer between Conv2D and Activation layers is used. If **"in place" InplaceABN is used, it will allow for a decrease in** memory consumption. Available options are **True, False, "inplace."**
- **decoder_attention_type** – Attention module used in the decoder of the model. Available options are **None** and **scse** (https://arxiv.org/abs/1808.08127).
- **in_channels** – Several input channels for the model, default is 3 (RGB images)
- **classes** – A number of classes for output mask (or you can think as a number of channels of output mask)
- **activation** –
- An activation function to apply after the final convolution layer. Available options are **"sigmoid"**, **"softmax"**, **"logsoftmax"**, **"tanh"**, **"identity"**,
- **callable** and **None**.
- Default is **None**
- **aux_params** –
- Dictionary with parameters of the auxiliary output (classification head). Auxiliary output is build on top of encoder if **aux_params** is not **None** (default). Supported params:
    - classes (int): A number of classes
    - pooling (str): One of "max", "avg". Default is "avg"
    - dropout (float): Dropout factor in [0, 1)
- **activation (str): An activation function to apply "sigmoid"/"softmax"**
    - (could be **None** to return logits)

Returns:

Unet

Return type:

torch.nn.Module

Use EfficientNet

Encoder	Weights	Params, M
efficientnet-b0	imagenet	4M

Use Losses:-

DiceLoss

*class*segmentation_models_pytorch.losses.DiceLoss(*mode*, *classes=None*, *log_loss=False*, *from_logits=True*, *smooth=0.0*, *ignore_index=None*, *eps=1e-07*)
Dice loss for image segmentation task. It supports binary, multiclass and multilabel cases
Parameters:

- **mode** – Loss mode 'binary', 'multiclass' or 'multilabel'
- **classes** – List of classes that contribute in loss computation. By default, all channels are included.
- **log_loss** – If True, loss computed as - log(dice_coeff), otherwise 1 - dice_coeff
- **from_logits** – If True, assumes input is raw logits
- **smooth** – Smoothness constant for dice coefficient (a)
- **ignore_index** – Label that indicates ignored pixels (does not contribute to loss)
- **eps** – A small epsilon for numerical stability to avoid zero division error (denominator will be always greater or equal to eps)

Shape

- y_pred - torch.Tensor of shape (N, C, H, W)
- y_true - torch.Tensor of shape (N, H, W) or (N, C, H, W)

SoftBCEWithLogitsLoss

*class*segmentation_models_pytorch.losses.SoftBCEWithLogitsLoss(*weight=None*, *ignore_index=-100*, *reduction='mean'*, *smooth_factor=None*, *pos_weight=None*)[source]
Drop-in replacement for torch.nn.BCEWithLogitsLoss with few additions: ignore_index and label_smoothing
Parameters:

- **ignore_index** – Specifies a target value that is ignored and does not contribute to the input gradient.
- **smooth_factor** – Factor to smooth target (e.g. if smooth_factor=0.1 then [1, 0, 1] -> [0.9, 0.1, 0.9])

Shape

- y_pred - torch.Tensor of shape NxCxHxW
- y_true - torch.Tensor of shape NxHxW or Nx1xHxW

IOU Score:-

The Intersection over Union (IOU) score, also known as the Jaccard Index, is a metric used to evaluate the accuracy of a segmentation model's predictions by measuring the overlap between the predicted segmentation mask and the ground truth mask. It is widely used in tasks involving object detection, instance segmentation, and semantic segmentation.

The IOU score is calculated as the ratio of the intersection area to the union area of the predicted and ground truth masks. Mathematically, it can be expressed as:

IOU = Area(Predicted ∩ Ground Truth) / Area(Predicted U cup Ground Truth)

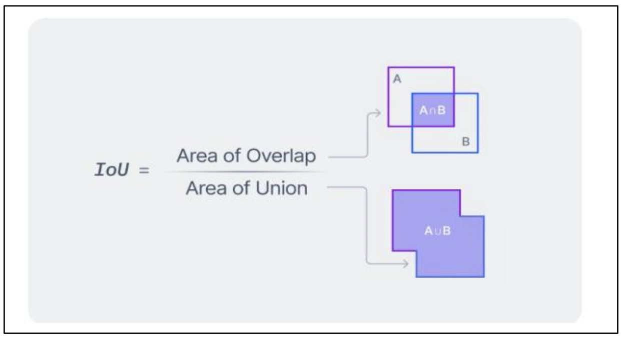
Where:

Area(Predicted ∩ Ground Truth) represents the area of overlap between the predicted and ground truth masks.
Area(Predicted U Ground Truth) represents the total area covered by both the predicted and ground truth masks.

$$Intersection\ over\ Union\ \left(IoU\right) = \frac{|\mathrm{A} \cap \mathrm{B}|}{|\mathrm{A}| \cup |\mathrm{B}|}$$

But for binary classification, it is written as:

$$Intersection\ over\ Union\ \left(IoU\right) = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN} + \mathrm{FP}}$$



The IOU score ranges from 0 to 1, where a score of 1 indicates perfect overlap between the predicted and ground truth masks, while a score of 0 indicates no overlap at all.

In the context of segmentation tasks, a higher IOU score indicates better agreement between the model's predictions and the ground truth labels, implying better segmentation accuracy.

Overall, the IOU score is a valuable metric for assessing the quality of segmentation models and is commonly used during evaluation to quantify their performance.

Documentation of segmentation model pytorch