Segmentation on Kvasir-SEG Report

Arad Vazirpanah-610399182

1 Dataset Familiarization

Kvasir dataset contains images of Gastrointestinal Polyp and their related masks.

As we can see the bboxes.json file, consists of some information about each data (polyp image).

- a dictionary belongs to each image.
- There is some information about the size of the image (as height and width).
- Also there is a bbox, which has an information about the mask image related to the original image.
- This bbox for each data, determines a rectangle that Specifies the region of interest within an image.

```
bboxes['cju0qkwl35piu0993l0dewei2']
```

```
{'height': 529,
 'width': 622,
 'bbox': [{'label': 'polyp', 'xmin': 38, 'ymin': 5, 'xmax': 430, 'ymax': 338}]}
```

As we can see, the details of the mask (white) part in the Mask image are available in the bbox in json file.

• y and x starts from upper-left corner. So the numbers in bbox totally make sense.

This is an example for original image and it's given mask. We are going to train some models so that they can predict these masks.

2 Preprocessing

2.1 Normalize and Resize:

Using function resize_normalize, we make all the images, same size. And also normalize them by dividing by 255.

1. get the image from given path.
2. Resize it to the target size (224×224 here).
3. Normalize the array, dividing by 255.

```
def resize_normalize(image_path, target_size=(128, 128), normalize=True,␣
↪mask=False):
    img = Image.open(image_path)
    if mask:
        img = (img.convert('L')).resize(target_size)
    else:
        img = img.resize(target_size)
    img_array = np.array(img)
    if normalize:
        img_array = img_array / 255.0

    return img_array
```

In this cell we resize and normalize each image and it's mask using resize_normalize function:

For example this is an images and it's related mask after preprocessing:

Resized-Normalized image

Resized-Normalized mask

3 Model Selection and Implementation

We use tersorflow to implement the U-net model. These are the blocks used in U-net implementation:

1. down_block:

- This function defines a down-sampling block used in a U-Net architecture.
- It takes input x, which represents the input tensor to the block.
- It applies two convolutional layers. The default activation function is ReLU.
- After each convolutional layer, it performs max pooling with a pool size of (2, 2) and a stride of (2, 2) to reduce the spatial dimensions.

2. up_block:

- This function defines an up-sampling block in a U-Net architecture.
- It takes input x, which represents the input tensor to the block, and skip, which represents the skip connection from the corresponding down-sampling block.
- It first applies up-sampling to the input tensor to double its spatial dimensions.
- It then concatenates the up-sampled tensor with the skip connection tensor to incorporate skip connections.
- After concatenation, it applies two convolutional layers with the specified parameters.

3. bottleneck:

- This function defines the bottleneck or central part of the U-Net architecture.
- It applies two convolutional layers.

```
def down_block(x, filters, kernel_size=(3, 3), padding="same", strides=1, activation="relu"):
    conv = keras.layers.Conv2D(filters, kernel_size, padding=padding, strides=strides,␣
 ↪activation=activation)(x)
    conv = keras.layers.Conv2D(filters, kernel_size, padding=padding, strides=strides,␣
 ↪activation=activation)(conv)
    pooling = keras.layers.MaxPool2D((2, 2), (2, 2))(conv)
    return conv, pooling


def up_block(x, skip, filters, kernel_size=(3, 3), padding="same", strides=1, activation="relu"):
    up_sampling = keras.layers.UpSampling2D((2, 2))(x)
    concat = keras.layers.Concatenate()([up_sampling, skip])
    conv = keras.layers.Conv2D(filters, kernel_size, padding=padding, strides=strides,␣
 ↪activation=activation)(concat)
    conv = keras.layers.Conv2D(filters, kernel_size, padding=padding, strides=strides,␣
 ↪activation=activation)(conv)
    return conv


def bottleneck(x, filters, kernel_size=(3, 3), padding="same", strides=1, activation="relu"):
    conv = keras.layers.Conv2D(filters, kernel_size, padding=padding, strides=strides,␣
 ↪activation=activation)(x)
    conv = keras.layers.Conv2D(filters, kernel_size, padding=padding, strides=strides,␣
 ↪activation=activation)(conv)
    return conv
```

3.1 U-net:

Here we define the first model which is a U-Net model. This is the architecture:

- It starts by defining an input layer with shape (128, 128, 3).
- Then, it applies three down-sampling blocks (down_block) with decreasing number of filters (8, 16, 32, 64, 128, 256).
- After the last down-sampling block, it applies the bottleneck operation to capture the most abstract features.
- Next, it applies three up-sampling blocks (up_block) with increasing number of filters (256, 128, 64, 32, 16, 8) and incorporates skip connections from the corresponding down-sampling blocks.
- Finally, it applies a convolutional layer with a single filter and a sigmoid activation function to generate the output mask.

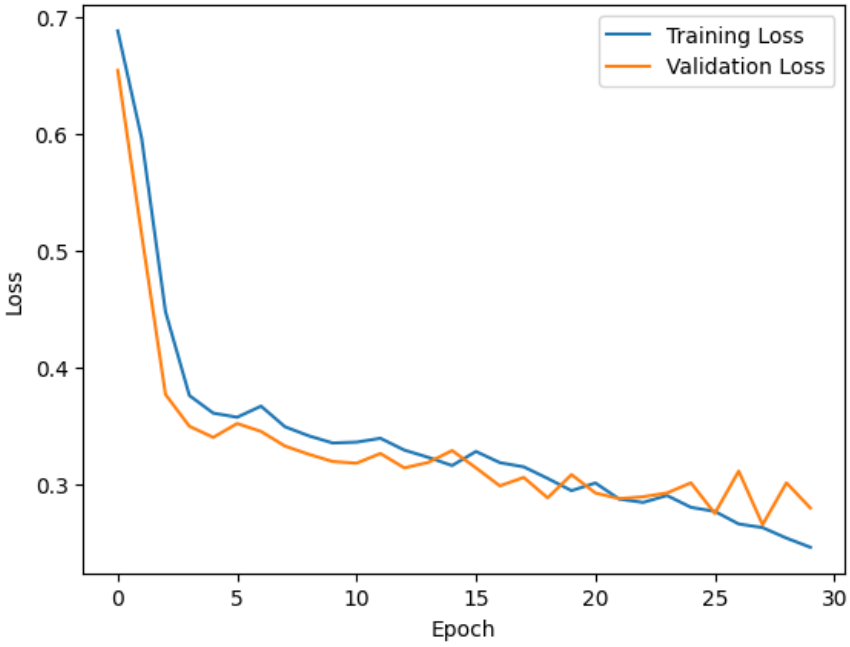
This model is used to segment the given polyp images and predict the masks. So we can determine if a patient have a polyp or not.

```
def UNet():
    pool0 = keras.Input((128, 128, 3))
    conv1, pool1 = down_block(pool0, 8)
    conv2, pool2 = down_block(pool1, 16)
    conv3, pool3 = down_block(pool2, 32)
    conv4, pool4 = down_block(pool3, 64)
    conv5, pool5 = down_block(pool4, 128)
    conv6, pool6 = down_block(pool5, 256)

    bn = bottleneck(pool6, 512)

    up1 = up_block(bn, conv6, 256)
    up2 = up_block(up1, conv5, 128)
    up3 = up_block(up2, conv4, 64)
    up4 = up_block(up3, conv3, 32)
    up5 = up_block(up4, conv2, 16)
    up6 = up_block(up5, conv1, 8)

    outputs = keras.layers.Conv2D(1, (1, 1), padding="same",␣
 ↪activation="sigmoid")(up6)
    model = keras.models.Model(pool0, outputs)
    return model

model = UNet()
model.compile(optimizer="adam", loss="binary_crossentropy", metrics=["acc"])
model.summary()
```

This is the accuracy on train and test on this model: The accuracy for both train and test data, are somehow equal and equivalent to 89.59%. Which is not a bad accuracy, but the important metric for comparing the models, is Dice coefficient or Intersection over Union (IoU) for segmentation task.

```
10/10 [==============================] - 2s 160ms/step - loss: 0.3743 - acc:
0.8959
Test Loss: 0.37429097294807434
Test Accuracy: 0.8958728313446045
```

And this is the history for loss and epoch. As we can see loss is decreasing for both train and validation set.



3.1.1 Dice score

This is the dice coefficient function:

- The Dice coefficient is a commonly used metric for evaluating the performance of segmentation algorithms.
- It measures the spatial overlap between two binary masks, typically a predicted mask and a ground truth mask.

How to calculate Dice Score:

1. Intersection: Calculate the number of overlapping pixels between the predicted mask P and the ground truth mask G.
2. Calculate the total number of pixels in both the predicted and ground truth masks.

3. Calculate the Dice coefficient using the formula:

$$\frac{2 \times |P \cap G|}{|P| + |G|}$$

```
                def dice_coefficient(y_true, y_pred):
                target_shape = (y_true.shape[0], y_true.shape[1])
                y_true = y_true.reshape(target_shape)
                y_pred = y_pred.reshape(target_shape)

                y_pred = np.where(y_pred < 0.5, 0, 1)
                return (np.sum(y_true == y_pred))/
↪(target_shape[0]*target_shape[1])
```

As we can see the Dice Score for testset on the first U-net model is 89.5%.

```
10/10 [==============================] - 2s 157ms/step
Average Dice Score: 0.895872802734375
```

3.1.2 IoU

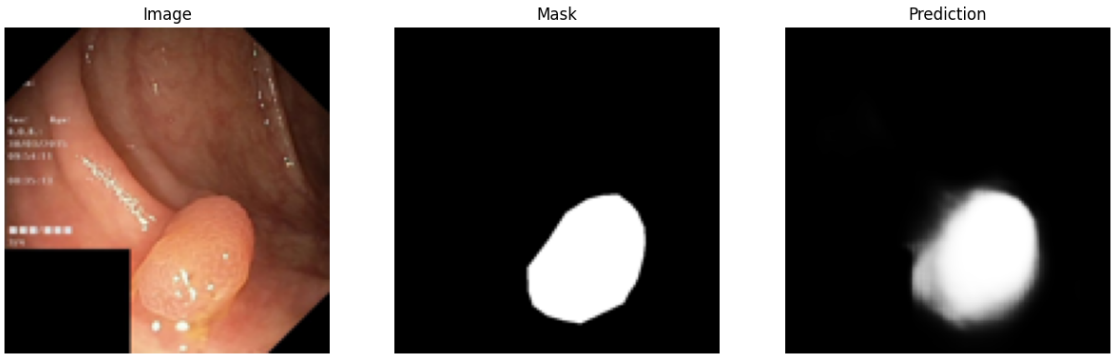
IoU provides a measure of how well the predicted segmentation mask overlaps with the ground truth mask.

- Calculate the intersection.
- Calculate the union.
- Divide intersection by union.

```
def calculate_iou(y_true, y_pred):
    intersection = np.logical_and(y_true, y_pred)
    union = np.logical_or(y_true, y_pred)
    iou_score = np.sum(intersection) / np.sum(union)
    return iou_score
```

```
Intersection over Union (IoU) score: 0.15464396158854166
```

These are the original image, the mask we want gain using the model and finally the mask we have earned by the first U-net model.



3.2 Stack U-net

This is the Stack U-Net architecture:

- First of all, the data is $256 \times 256 \times 4$, because the output of the first U-net is concatenated with the original image.
- Downsampling Path (First UNet): Sequentially applies downsampling blocks (down_block) to extract features at different scales. Each downsampling block reduces the spatial dimensions while increasing the number of channels.
- Bottleneck Layer (First UNet): Applies a bottleneck layer to capture high-level features.
- Upsampling Path (First UNet): Applies upsampling blocks (up_block) to reconstruct the original image size from the extracted features. Each upsampling block increases the spatial dimensions while reducing the number of channels.
- Output Layer (First UNet): Applies a convolutional layer to generate the first set of outputs with a sigmoid activation function.
- Concatenation: Concatenates the original input with the outputs of first U-net.
- Downsampling Path (Second UNet): Applies downsampling blocks to the concatenated features to extract additional features.
- Bottleneck Layer (Second UNet): Applies a bottleneck layer to capture high-level features from the concatenated features.
- Upsampling Path (Second UNet): Applies upsampling blocks to reconstruct the original image size from the concatenated features.
- Output Layer (Second UNet): Applies a convolutional layer to generate the final set of outputs with a sigmoid activation function.

```
def stack_UNet():

        pool0 = keras.Input((128, 128, 3))
        # conv1, pool1 = down_block(pool0, 8)
        conv2, pool2 = down_block(pool0, 16)
        conv3, pool3 = down_block(pool2, 32)
        conv4, pool4 = down_block(pool3, 64)
        # conv5, pool5 = down_block(pool4, 128)
        # conv6, pool6 = down_block(pool5, 128)
        # conv7, pool7 = down_block(pool6, 256)

        bn = bottleneck(pool4, 128)

        # up1 = up_block(bn, conv5, 128)
        up2 = up_block(bn, conv4, 64)
        up3 = up_block(up2, conv3, 32)
        up4 = up_block(up3, conv2, 16)
        # up5 = up_block(up4, conv1, 8)
        # up6 = up_block(up5, conv2, 8)
```

```
        # up7 = up_block(up6, conv1, 4)

        first_outputs = keras.layers.Conv2D(1, (1, 1), padding="same",␣
↪activation="sigmoid")(up4)

        concate = keras.layers.Concatenate()([pool0, first_outputs])
        # conv2_1, pool2_1 = down_block(concate, 8)
        # conv2_2, pool2_2 = down_block(pool2_1, 16)
        conv2_3, pool2_3 = down_block(concate, 32)
        conv2_4, pool2_4 = down_block(pool2_3, 64)
        conv2_5, pool2_5 = down_block(pool2_4, 128)
        # conv2_6, pool2_6 = down_block(pool2_5, 128)
        # conv2_7, pool2_7 = down_block(pool2_6, 256)

        bn = bottleneck(pool2_5, 256)

        up2_1 = up_block(bn, conv2_5, 128, activation="sigmoid")
        up2_2 = up_block(up2_1, conv2_4, 64, activation="sigmoid")
        up2_3 = up_block(up2_2, conv2_3, 32, activation="sigmoid")
        # up2_4 = up_block(up2_3, conv2_2, 16, activation="sigmoid")
        # up2_5 = up_block(up2_4, conv2_1, 8, activation="sigmoid")
        # up2_6 = up_block(up2_5, conv2_2, 8, activation="sigmoid")
        # up2_7 = up_block(up2_6, conv2_1, 4, activation="sigmoid")

        final_outputs = keras.layers.Conv2D(1, (1, 1), padding="same",␣
↪activation="sigmoid")(up2_3)

        model = keras.models.Model(pool0, final_outputs)
        return model

        model2 = stack_UNet()
        model2.compile(optimizer="adam", loss="binary_crossentropy",␣
↪metrics=["acc"])
        model2.summary()
```

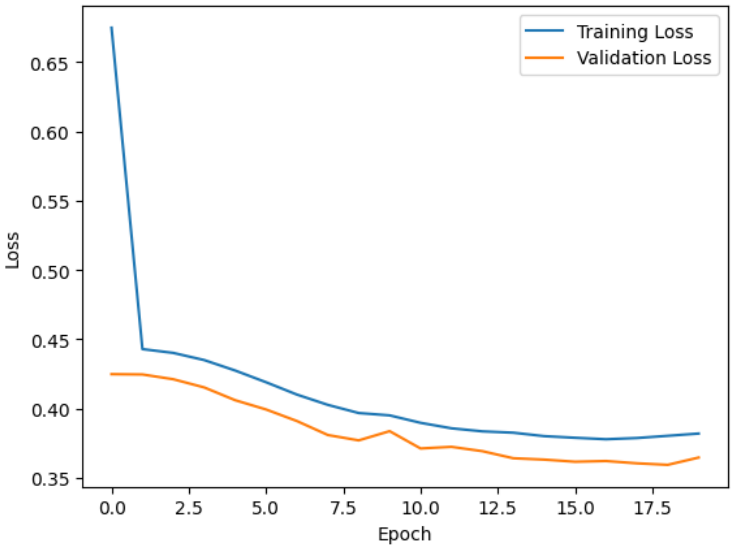
As we can see, the accuracy for this model on train data is 83.3% and for validation set is 84.2%.

```
Epoch 20/20
19/19 [==============================] - 179s 9s/step - loss: 0.3819 - accuracy:␣
 ↪0.8338 - val_loss: 0.3646 - val_accuracy: 0.8429
```

This is the accuracy on test set:

```
10/10 [==============================] - 3s 327ms/step - loss: 0.4118 -
accuracy: 0.8454
Test Loss: 0.4118417799472809
Test Accuracy: 0.8453560471534729
```

This is the history for loss vs epoch in stack U-net training.

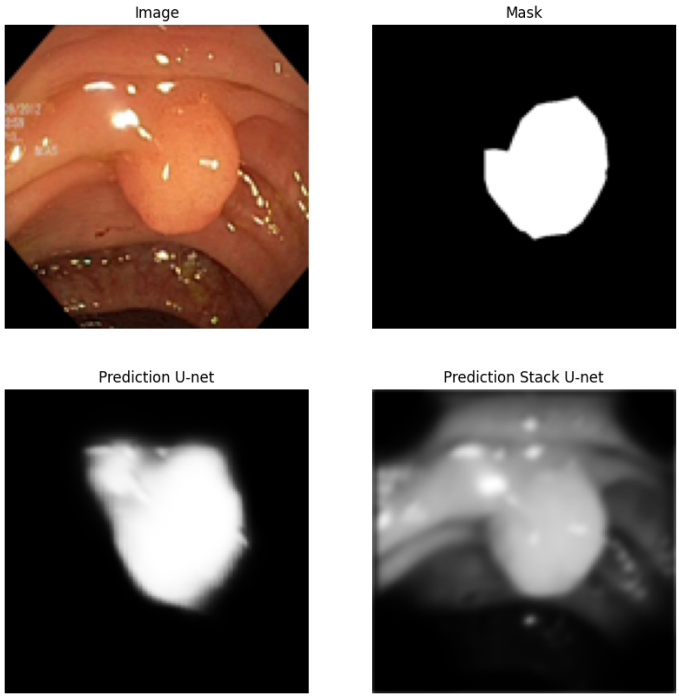


3.2.1 Dice Score

```
10/10 [==============================] - 3s 300ms/step
Average Dice Score: 0.8453560384114583
```

3.2.2 IoU

```
Intersection over Union (IoU) score: 0.15464396158854166
```



4 Conclusion

We have trained the dataset on two models:

1. single U-net
2. stack U-net

As we have seen, the Dice Score and the IoU score, was better and higher for single U-net. But the model that has been trained as stack U-net, wasn't so deep, so the results was that the single model is predicting the polyps, better than the stack U-net.

But we know that if the stack U-net's layers were defined better and there were more layers, for sure this model would be better.

The idea used for the stacked U-Net is genius, when concatenating the input image and the mask gained by the first part of the network (first U-net), makes the second U-net to have more attention on the segmented regions. As a result, the segmentation would be much more accurate and clear.

The attention map helps the prediction a lot. The model will help a lot of patients, so that their gastrointestinal polyps can be find more accurately and the cancer would be recognized faster.