

Artificial Intelligence Project Report

Image Segmentation

Dr.Sajedi

Author: Morteza Rashidkhan

1. Preprocessing:

در ابتدا ما عکس ها و mask ها را از فایل می خوانیم و آن ها را resize می کنیم به عکس های با سایز 256*256 و همچنین آن ها را normalize میکنیم و آن ها را برای training آماده می کنیم.

```
class TfdatasetPipeline:
    ...
    ...
    def __read_image_and_mask(self, image_path: str, mask_path: str) -
    >                                     tf.Tensor:

        img_raw          =          tf.io.read_file(image_path)
        mask_raw          =          tf.io.read_file(mask_path)

        img              =          tf.io.decode_jpeg(img_raw)
        mask              =          tf.io.decode_jpeg(mask_raw, channels=1)

        img = tf.image.convert_image_dtype(img, tf.float32) # normalize
the          values          between          0-1

        mask = tf.image.convert_image_dtype(mask, tf.float32) #
normalize          the          values          between          0-1

        img = tf.image.resize(img, [self.IMG_H, self.IMG_W])
        mask = tf.image.resize(mask, [self.IMG_H, self.IMG_W])

        return img, mask
```

2. Model Selection and Implementation:

2.1. Selected model: PraNet (Parallel Reverse Attention Network)

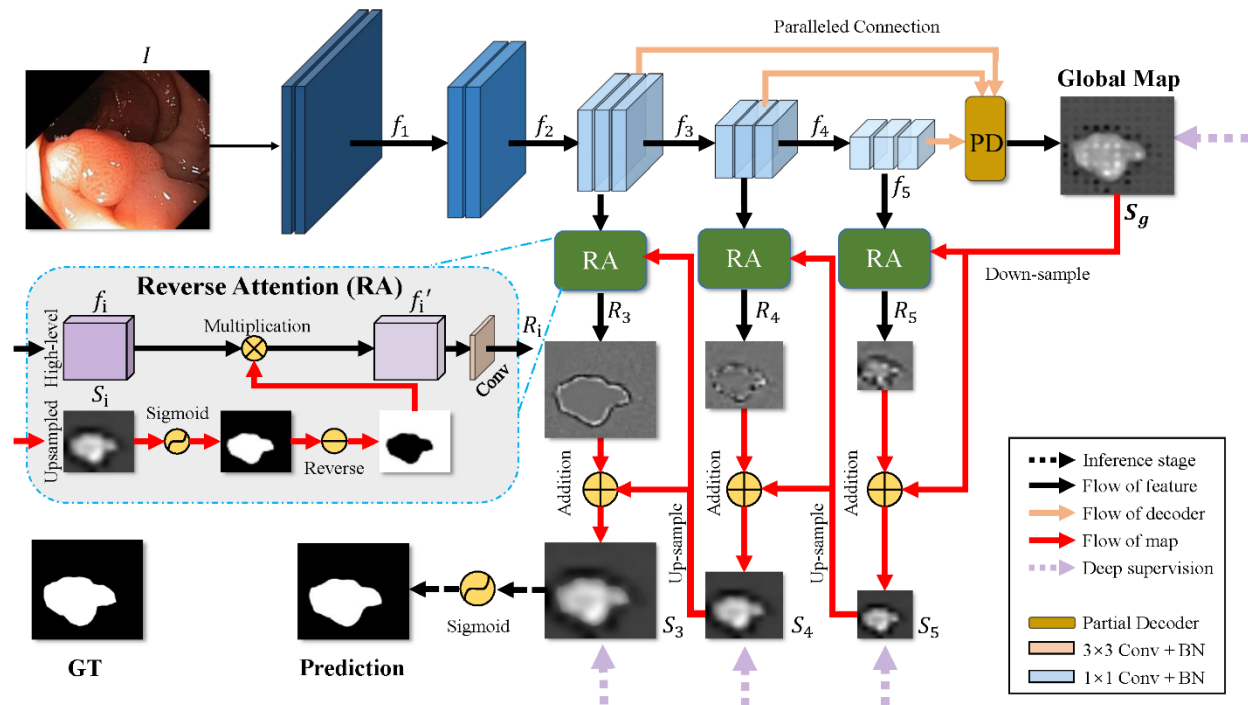
این مدل که در مقاله نیز به آن اشاره شده 3 تا مولفه اصلی دارد که به شرح زیر است:

- Feature Extractor Backbone
- Parallel Partial Decoder Block
- Reverse Attention Block (Reverse Attention for Salient object detection)

Feature Extractor Backbone:

یک Feature Extractor Backbone می تواند هر مدل شناخت تصویر با کارایی بالا باشد. برای خروجی با دقت بالا از resnet50 که روی داده های Imagenet از قبل train شده است. هدف نهایی استفاده از یک

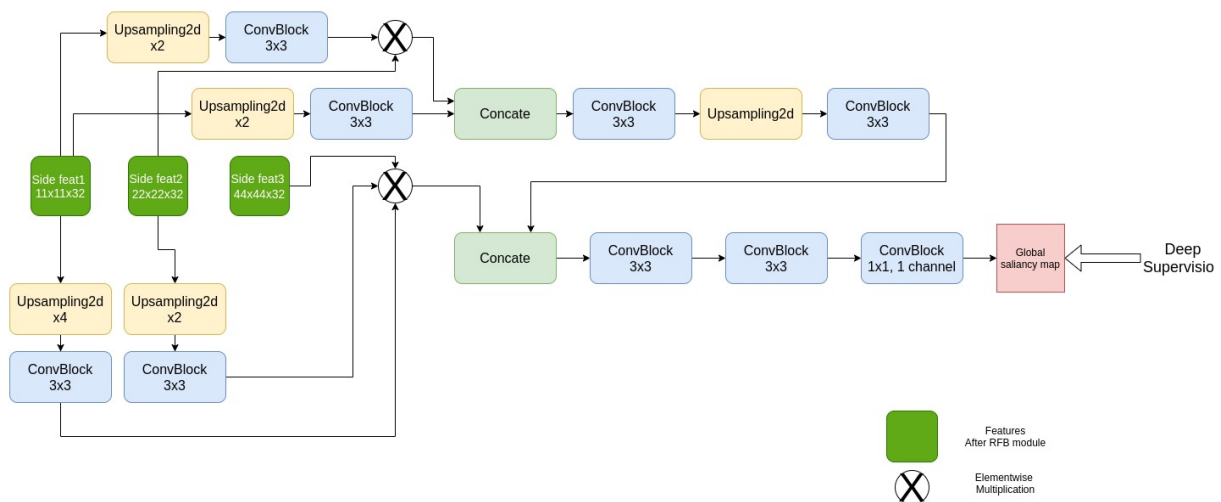
Feature Extractor Backbone پیش‌آموزش دیده، استخراج ویژگی‌های سطح بالا از تصاویر و جمع‌آوری این ویژگی‌ها به منظور به‌دست آوردن یک نقشه سراسری است.



PraNet Architecture

Parallel Partial Decoder:

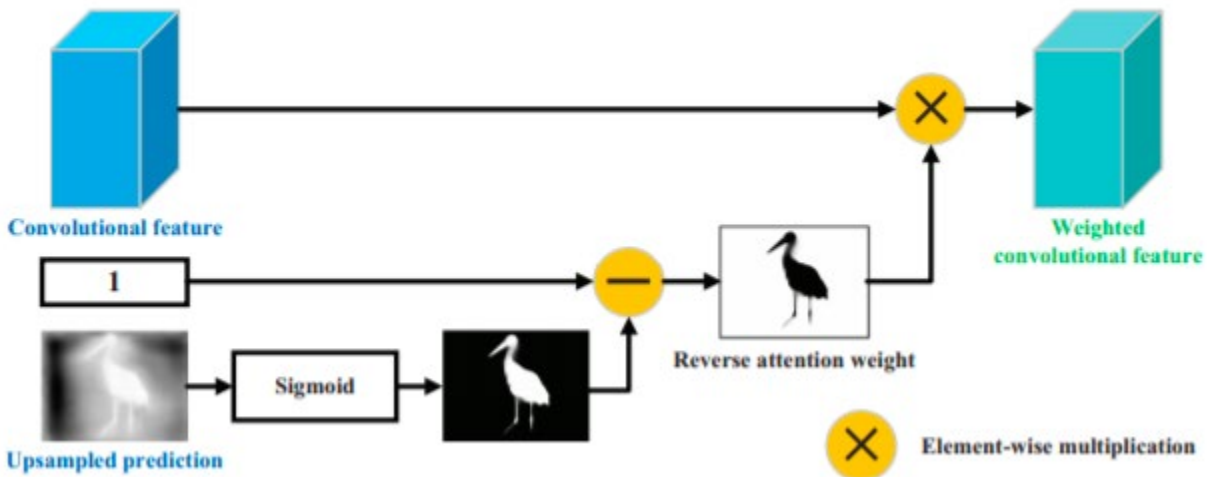
بیشتر مدل‌های تقسیم تصویر پزشکی محبوب مانند Unet, U-net++, ResUnet و غیره، مدل‌های کامل کدگشا هستند یعنی تمام ویژگی‌های چند سطحی از شبکه اصلی را جمع‌آوری می‌کنند. اما در مقایسه با ویژگی‌های سطح بالای اسکلت، ویژگی‌های سطح پایین به دلیل داشتن وضوح فضای بزرگتر، نیازمند منابع محاسباتی بیشتری هستند، اما کمتر به عملکرد کلی مدل ارتباط دارند. بنابراین، این مدل از یک بلوک PPD (Parallel Partial Decoder) استفاده می‌کند که تنها ویژگی‌های سطح بالا که از مدل پیش‌آموزش دیده (Global Saliency Map) resnet50 استخراج شده‌اند را جمع‌آوری می‌کند تا یک نقشه جهت سراسری (Global Saliency Map) به‌دست آید. این بلوک تمام ویژگی‌های سطح بالا را به صورت موازی جمع می‌کند و اینطوری مدل سریعتر و کارآمدتر می‌شود.



PPD block

Reverse Attention:

در یک محیط بالینی، پزشکان ابتدا به طور تقریبی ناحیه پولیپ را تعیین می کنند و سپس بافت های محلی را به دقت بررسی می کنند تا به طور دقیق پولیپ را برجسب گذاری کنند. نقشه جهت سراسری ما از عمیق ترین لایه CNN مشتق شده است، که فقط می تواند یک مکان نسبتاً ناهموار از بافت های پولیپ را بدون جزئیات ساختاری ثبت کند (شکل معماری پیشنهادی را ببینید). در این مدل یک استراتژی اصلی برای استخراج تدریجی نواحی پولیپ که متمایز از روش پاک کردن شی پیش زمینه همانطور که در مقاله Reverse Attention برای تشخیص شی برجسته ارائه شده است، استفاده می شود. در اینجا به جای تجمیع ویژگی ها از همه سطوح مانند ++Unet U-net و ResUnet، یادگیری تطبیقی Reverse Attention را در سه ویژگی سطح بالا موازی پیشنهاد می کند. به عبارت دیگر، معماری ما می تواند به طور متوالی مناطق و جزئیات مکمل را با پاک کردن مناطق پولیپ تخمین زده شده از ویژگی های خروجی جانبی سطح بالا استخراج کند، جایی که تخمین موجود از لایه عمیق تر نمونه برداری می شود.



Reverse Attention block

2.2.Implementation:

```
class PRANet(tf.keras.Model):
    def __init__(self, IMG_H: int = 352, IMG_W: int = 352, filters: int
= 32, backbone_arch:str = 'resnet50', backbone_trainable: bool = True,
**kwargs):
        super(PRANet,
                self).__init__(**kwargs)
        self.IMG_H
            = IMG_H
        self.IMG_W
            = IMG_W
        self.filters
            = filters
        self.backbone_arc
            = backbone_arch
        self.backbone_trainable
            = backbone_trainable

        # pretrained resnet
        self.fe_backbone
            =
FE_backbone(model_architecture=self.backbone_arc,inshape=(self.IMG_H,
self.IMG_W,
3),
            is_trainable=self.backbone_trainable
            )
        self.backbone_feature_extractor
            =
self.fe_backbone.get_fe_backbone()

        # Receptive field blocks
        # 3 blocks for three high level features from resnet
        self.rfb_2
            = RFB(filters=self.filters,
            name="rfb_2")
        self.rfb_3
            = RFB(filters=self.filters,
            name="rfb_3")
        self.rfb_4
            = RFB(filters=self.filters,
            name="rfb_4")

        # Paraller Partial Decoder Block
        self.ppd
            = PartialDecoder(filters=self.filters,
name="partial_decoder")
        self.resize_sg = preprocessing.Resizing(self.IMG_H, self.IMG_W,
```

```

name='salient_out_5')

#           reverse           attention           branch           4
self.resize_4 = preprocessing.Resizing(self.IMG_H//32,
self.IMG_W//32, name="resize4")
self.ra_4 = ReverseAttention(
    filters=256, kernel_size=(5, 5), branch='gsmap',
name="reverse_attention_br4")
self.resize_s4 = preprocessing.Resizing(self.IMG_H,
self.IMG_W, name="salient_out_4")

#           reverse           attention           branch           3
self.resize_3 = preprocessing.Resizing(self.IMG_H//16,
self.IMG_W//16, name="resize3")
self.ra_3 = ReverseAttention(filters=64,
name="reverse_attention_br3")
self.resize_s3 = preprocessing.Resizing(self.IMG_H, self.IMG_W,
name="salient_out_3")

#           reverse           attention           branch           2
self.resize_2 = preprocessing.Resizing(self.IMG_H//8,
self.IMG_W//8, name="resize2")
self.ra_2 = ReverseAttention(filters=64,
name="reverse_attention_br2")
self.resize_s2 = preprocessing.Resizing(self.IMG_H, self.IMG_W,
name="final_salient_out_2")

def call(self, x: tf.Tensor):
    ...

    return lateral_out_sg, lateral_out_s4, lateral_out_s3,
lateral_out_s2

    ...

    def get_config(self):
        ...
    @classmethod
    def from_config(cls, config):
        return cls(**config)

    def build_graph(self, inshape:tuple) -> tf.keras.Model:
        x = tf.keras.layers.Input(shape=inshape)

```

```
return tf.keras.Model(inputs=[x], outputs = self.call(x),
name='PRAnet')
```

کد کامل داخل در مسیر model/PRA_net.py موجود است.

3. Training and Validation:

ما از 0.9 داده ها برای train و validation استفاده کردیم و 0.1 باقی مانده را برای prediction نگه داشتیم. برای train کردن تعداد epoch ها را 25 انتخاب کردیم و همچنین تعداد batch ها 8 در نظر گرفتیم و learning rate را برای شروع 0.001 گذاشتیم. همچنین برای optimizer از Adam استفاده کردیم. تابع Loss که انتخاب کردیم در واقع مجموع Weighted BCE(Binary Cross Entropy loss و Weighted Dice loss است.

$$\text{loss} = \text{WBCE loss} + \text{WDice loss}$$

کد مربوط به train این مدل در فایل PraNet_ARC_Final.ipynb قرار دارد.

در ادامه قسمتی از خروجی موقع training مشاهده می کنیم:

```
epoch: 8 - loss: 0.7985283136367798 - dice: 0.9258408546447754 - IoU: 0.8636079430580139 - val_loss: 5.030370712280273 - val_dice: 0.21117088198661804 - val_IoU: 0.1719688177108764

epoch: 9 - loss: 0.9285917282104492 - dice: 0.9176560640335083 - IoU: 0.8489720821380615 - val_loss: 3.555083751678467 - val_dice: 0.5403826832771301 - val_IoU: 0.4207243323326111

epoch: 10 - loss: 0.9750457406044006 - dice: 0.9102895259857178 - IoU: 0.8404217958450317 - val_loss: 2.309605836868286 - val_dice: 0.6753308176994324 - val_IoU: 0.6168214082717896

epoch: 11 - loss: 0.7543209791183472 - dice: 0.9283473491668701 - IoU: 0.867800235748291 - val_loss: 1.5406849384307861 - val_dice: 0.8341243267059326 - val_IoU: 0.7482954263687134

epoch: 12 - loss: 0.6943879127502441 - dice: 0.9375870227813721 - IoU: 0.8831126689910889 - val_loss: 1.5587424039840698 - val_dice: 0.8188931941986084 - val_IoU: 0.7396512031555176

epoch: 13 - loss: 0.6155073642730713 - dice: 0.9447202682495117 - IoU: 0.8955438137054443 - val_loss: 1.6553741693496704 - val_dice: 0.7905910015106201 - val_IoU: 0.6742314100265503

epoch: 14 - loss: 0.6066497564315796 - dice: 0.9469993114471436 - IoU: 0.8996536135673523 - val_loss: 1.8086917400360107 - val_dice: 0.8421146273612976 - val_IoU: 0.7384793758392334

epoch: 15 - loss: 0.566040575504303 - dice: 0.9533501863479614 - IoU: 0.9110949039459229 - val_loss: 1.790685772895813 - val_dice: 0.8135406970977783 - val_IoU: 0.7212449312210083
```

```

epoch: 16 - loss: 0.5662965774536133 - dice: 0.9509921073913574 - IoU: 0.9068719744682312 - val_loss: 1.307141661643982 - val_dice: 0.8455063700675964 - val_IoU: 0.7430897355079651

epoch: 17 - loss: 0.546355664730072 - dice: 0.9552727341651917 - IoU: 0.9147423505783081 - val_loss: 1.8528587818145752 - val_dice: 0.8191965818405151 - val_IoU: 0.714682400226593

epoch: 18 - loss: 0.5117009878158569 - dice: 0.9602827429771423 - IoU: 0.9239192008972168 - val_loss: 1.1892342567443848 - val_dice: 0.8684498071670532 - val_IoU: 0.788508415222168

epoch: 19 - loss: 0.4557315707206726 - dice: 0.9663265943527222 - IoU: 0.9349252581596375 - val_loss: 1.0484495162963867 - val_dice: 0.8760969638824463 - val_IoU: 0.8056252002716064

epoch: 20 - loss: 0.42226889729499817 - dice: 0.9702930450439453 - IoU: 0.9424219131469727 - val_loss: 1.036953091621399 - val_dice: 0.8837442994117737 - val_IoU: 0.8137525320053101

epoch: 21 - loss: 0.40440744161605835 - dice: 0.9708181619644165 - IoU: 0.9433499574661255 - val_loss: 1.1434648036956787 - val_dice: 0.8760643005371094 - val_IoU: 0.799681127071380

epoch: 22 - loss: 0.3928317129611969 - dice: 0.9729300737380981 - IoU: 0.9473636150360107 - val_loss: 1.0129520893096924 - val_dice: 0.8865821361541748 - val_IoU: 0.8120559453964233

epoch: 23 - loss: 0.41227203607559204 - dice: 0.9648968577384949 - IoU: 0.9323859214782715 - val_loss: 1.16818368434906 - val_dice: 0.872435450553894 - val_IoU: 0.7921082973480225

epoch: 24 - loss: 0.38075557351112366 - dice: 0.9719101786613464 - IoU: 0.9454149007797241 - val_loss: 1.0940110683441162 - val_dice: 0.8732898235321045 - val_IoU: 0.790854275226593

epoch: 25 - loss: 0.44279852509498596 - dice: 0.9673148989677429 - IoU: 0.9368215203285217 - val_loss: 1.2860968112945557 - val_dice: 0.8664425611495972 - val_IoU: 0.790251016616821

```

4. Results and Analysis:

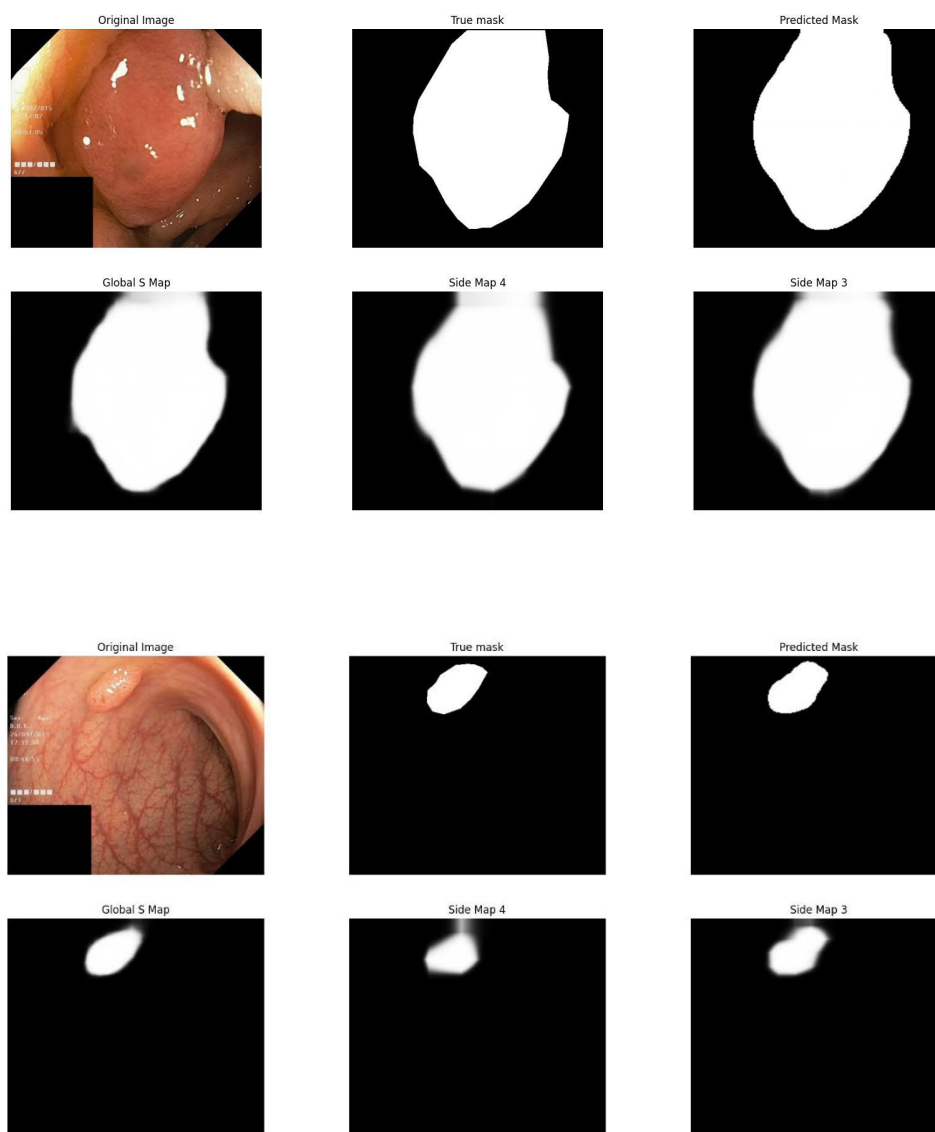
4.1. Performance

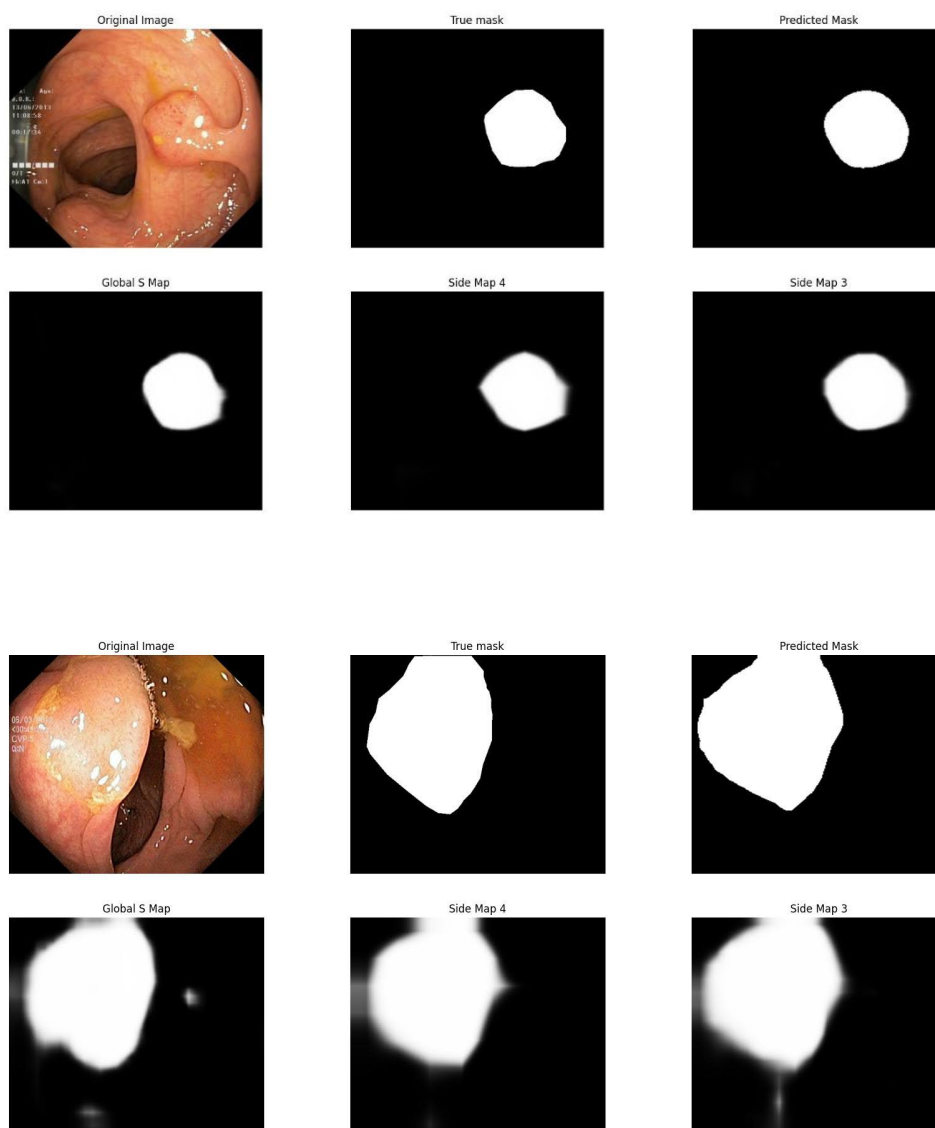
Architecture	Learning Rate	Epoch	Batch Size	DICE	IOU	MAE
PraNet_resnet50	1e-3	25	8	84.83	76.55	5.58
DPN68×2	1e-2	25	4	91.70	86.74	2.68

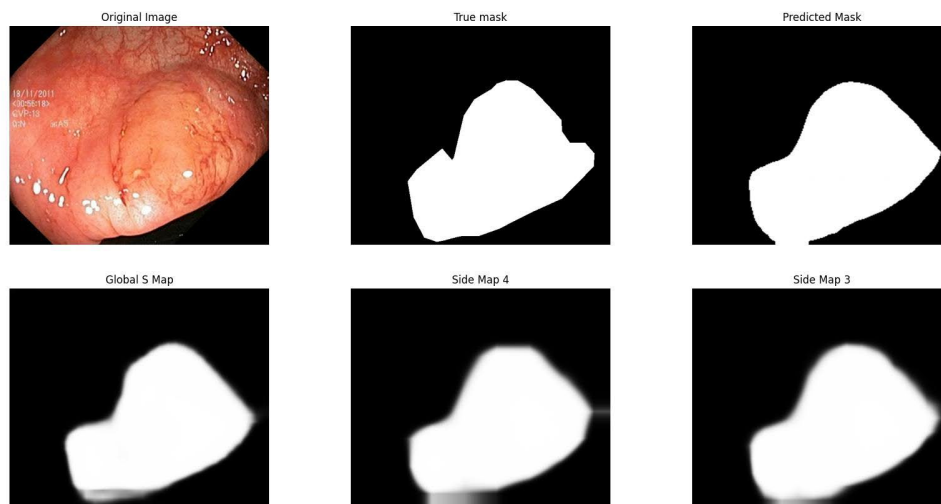
همانطور که از نتایج مشخص است مدلی که مقاله معرفی کرده یعنی DPN68×2 از نظر عملکرد کمی از مدل ما بهتر است.

در ادامه تعدادی از تصاویر که mask آنها توسط مدل train شده ما predict شده قرار داده شده است.

4.2.Results







4.3.Challenges faced during training

ابتدا مدل ساده UNet را انتخاب کرده بودم (که کد آن در فایل UNET_ARC_Older.ipynb موجود است) ولی نتایج آن خیلی خوب نبود و مقدار IOU حدود 33.00 بود و اما مشکلی که موقع train کردن این مدل به آن برخوردم این بود که loss آن به سرعت منفی می شد و تا بی نهایت می رفت. پس از کلی تست کردن متوجه شدم که مشکل از normalize کردن mask ها در مرحله preprocessing بود که در واقع باید mask را تقسیم بر 255 می کردم تا همه مقادیر بین 0 و 1 قرار بگیرند و من اینکار را انجام نمی دادم.