INTERNSHIP
ARCHITECTURE -3
MULTI RESUNET
REPORT

BY:

NAME: VANGA KANISHKA NADH

ROLL: 201CS165

EMAIL: vangakanishkanadh.201cs165@nitk.edu.in

MENOTR:

PROF. JENY RAJAN

Introduction:

The Multi-ResUNet algorithm is a deep learning architecture designed for semantic segmentation tasks, particularly in medical image analysis. It extends the U-Net architecture by incorporating multi-resolution pathways to capture fine-grained details while maintaining contextual information. Multi-ResUNet is known for its effectiveness in accurately segmenting objects or regions of interest within complex images.

Key Principles:

1. U-Net Architecture: Multi-ResUNet is built upon the U-Net architecture, consisting of an encoder and a decoder pathway. The encoder extracts high-level features through downsampling, while the decoder upsamples the features to generate segmentation masks.
2. Multi-Resolution Pathways: Multi-ResUNet employs multiple parallel pathways at different resolutions to capture features at various scales. This allows the algorithm to handle local details and global context effectively.

Relevant Papers:

1. Alom, M. Z., Taha, T. M., Yakopcic, C., Westberg, S., Sidike, P., Nasrin, M. S., ... & Asari, V. K. (2019). Recurrent Residual U-Net for Medical Image Segmentation. arXiv:1802.06955.

Dataset Description:

The dataset used for evaluation is the "2018 Data Science Bowl - Processed" cell nuclei segmentation dataset. It was sourced from Kaggle and is specifically curated for the task of segmenting cell nuclei in microscopic images. The dataset is widely used in the field of biomedical image analysis and has been employed to benchmark various segmentation algorithms.

670 Images 670 Masks = 1340 Total Images

Link: https://www.kaggle.com/datasets/84613660e1f97d3b23a89deb1ae6199a0c795ec1f31e2934527a7f7aad7d8c37

Model Architecture:

The Multi-ResUNet model architecture consists of an encoder pathway, a bridge layer, and a decoder pathway. It follows a U-Net-like structure with skip connections and introduces multi-resolution blocks for enhanced segmentation performance.

Key Components:

1. **Convolutional Block:**
   - Function: This block performs a 2D convolution operation followed by batch normalization and activation.
   - Components:
     - Conv2D: Performs a convolution operation on the input tensor.
     - BatchNormalization: Normalizes the outputs of the convolutional layer to stabilize training.
     - Activation: Applies the ReLU activation function to introduce non-linearity.
2. **Multi-Resolution Block**:
   - Function: This block captures multi-scale features by combining feature maps from different resolution pathways.
   - Components:

    - Convolutional Blocks: Sequential convolutional blocks with different filter sizes.
    - Concatenate: Concatenates the outputs of convolutional blocks.
    - BatchNormalization: Normalizes the concatenated feature maps.

3. **Residual Path:**

- Function: This block adds residual connections to the multi-resolution block for improved information flow.
- **Components:**
    - Convolutional Blocks: Sequential convolutional blocks with the same filter size.
    - Addition: Element-wise addition of the output from the convolutional block and the skip connection.
    - BatchNormalization: Normalizes the output of the addition operation.

4. **Encoder Block:**

- Function: This block consists of a multi-resolution block followed by a residual path and max pooling.
- Components:
    - Multi-Resolution Block: Captures multi-scale features.
    - Residual Path: Adds residual connections.
    - MaxPooling2D: Performs downsampling by selecting the maximum value within a specific region.

5. **Decoder Block:**

- Function: This block performs upsampling using transposed convolutions, concatenates skip connections, and applies the multi-resolution block.
- Components:
    - Conv2DTranspose: Performs transposed convolution for upsampling.
    - Concatenate: Concatenates the upsampled feature maps with the corresponding skip connection.
    - Multi-Resolution Block: Captures multi-scale features.

6. **Bridge:**

- Function: This layer connects the encoder and decoder pathways to preserve high-level features.
- Components:
    - Multi-Resolution Block: Captures multi-scale features.

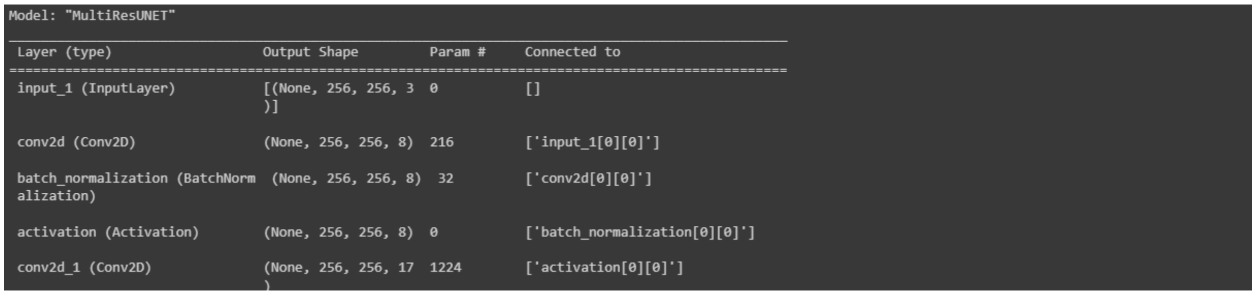
7. **Output:**

- Function: This block applies a 1x1 convolution followed by sigmoid activation to produce the final segmentation output.

```
from google.colab import drive
drive.mount('/content/drive/')
```

```
from tensorflow.keras.layers import Conv2D, BatchNormalization,
Activation, MaxPooling2D, Conv2DTranspose
from tensorflow.keras.layers import Concatenate, Input
from tensorflow.keras.models import Model

def conv_block(x, num_filters, kernel_size, padding="same", act=True):
    x = Conv2D(num_filters, kernel_size, padding=padding, use_bias=False)
(x)
    x = BatchNormalization()(x)
    if act:
        x = Activation("relu")(x)
    return x

def multires_block(x, num_filters, alpha=1.67):
    W = num_filters * alpha

    x0 = x
    x1 = conv_block(x0, int(W*0.167), 3)
    x2 = conv_block(x1, int(W*0.333), 3)
    x3 = conv_block(x2, int(W*0.5), 3)
    xc = Concatenate()([x1, x2, x3])
    xc = BatchNormalization()(xc)

    nf = int(W*0.167) + int(W*0.333) + int(W*0.5)
    sc = conv_block(x0, nf, 1, act=False)

    x = Activation("relu")(xc + sc)
    x = BatchNormalization()(x)
    return x
```

```python
def res_path(x, num_filters, length):
    for i in range(length):
        x0 = x
        x1 = conv_block(x0, num_filters, 3, act=False)
        sc = conv_block(x0, num_filters, 1, act=False)
        x = Activation("relu")(x1 + sc)
        x = BatchNormalization()(x)
    return x

def encoder_block(x, num_filters, length):
    x = multires_block(x, num_filters)
    s = res_path(x, num_filters, length)
    p = MaxPooling2D((2, 2))(x)
    return s, p

def decoder_block(x, skip, num_filters):
    x = Conv2DTranspose(num_filters, 2, strides=2, padding="same")(x)
    x = Concatenate()([x, skip])
    x = multires_block(x, num_filters)
    return x

def build_multiresunet(shape):
    """ Input """
    inputs = Input(shape)

    """ Encoder """
    p0 = inputs
    s1, p1 = encoder_block(p0, 32, 4)
    s2, p2 = encoder_block(p1, 64, 3)
    s3, p3 = encoder_block(p2, 128, 2)
    s4, p4 = encoder_block(p3, 256, 1)

    """ Bridge """
    b1 = multires_block(p4, 512)

    """ Decoder """
    d1 = decoder_block(b1, s4, 256)
    d2 = decoder_block(d1, s3, 128)
    d3 = decoder_block(d2, s2, 64)
    d4 = decoder_block(d3, s1, 32)

    """ Output """
    outputs = Conv2D(1, 1, padding="same", activation="sigmoid")(d4)
    """ Model """
    model = Model(inputs, outputs, name="MultiResUNET")

    return model

if __name__ == "__main__":
    shape = (256, 256, 3)
    model = build_multiresunet(shape)
    model.summary()
```



```
Model: "MultiResUNET"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_1 (InputLayer)           [(None, 256, 256, 3  0           []
                                )]

 conv2d (Conv2D)                (None, 256, 256, 8)  216         ['input_1[0][0]']

 batch_normalization (BatchNorm  (None, 256, 256, 8)  32         ['conv2d[0][0]']
 alization)

 activation (Activation)        (None, 256, 256, 8)  0           ['batch_normalization[0][0]']

 conv2d_1 (Conv2D)              (None, 256, 256, 17  1224        ['activation[0][0]']
                                )
```

Training Methodology and Parameters:

1. Optimization Algorithm: The optimization algorithm employed in the training process is Adam. It is a popular gradient-based optimization algorithm that adapts the learning rate for each parameter based on the first and second moments of the gradients.

2. Training Process:

   - The dataset is loaded and split into training, validation, and testing sets.
   - The images and corresponding masks are preprocessed, including resizing and normalization.
   - The training and validation datasets are created using the tf_dataset() function, which applies the preprocessing steps and batches the data.
   - The Multi-ResUNet model is built using the build_multiresunet() function.
   - The model is compiled with the binary cross-entropy loss function and the Adam optimizer with a learning rate of lr.
   - Several callbacks are set up to monitor the training progress and save the best model weights, reduce the learning rate on plateau, log metrics to a CSV file, and early stop if the validation loss does not improve.
   - The model is trained using the fit() function, specifying the number of epochs, training and validation datasets, and the steps per epoch and validation steps.

Training Parameters:

- Batch Size: 8
- Learning Rate (lr): 1e-4
- Number of Epochs: 10

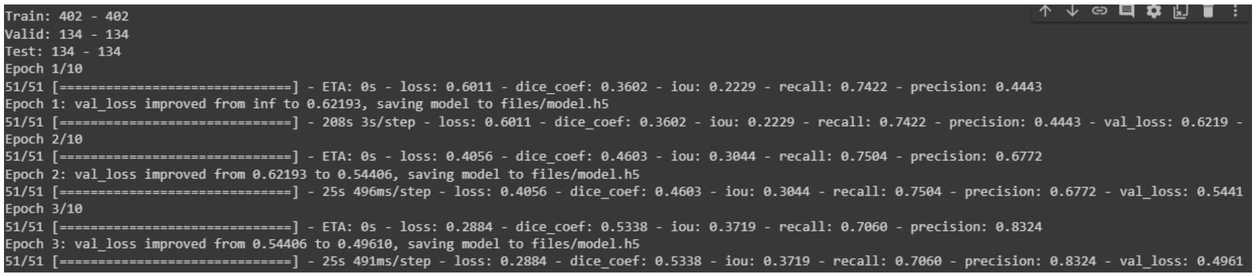
Optimization Algorithm: Adam

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import backend as K

def iou(y_true, y_pred):
    def f(y_true, y_pred):
        intersection = (y_true * y_pred).sum()
        union = y_true.sum() + y_pred.sum() - intersection
        x = (intersection + 1e-15) / (union + 1e-15)
        x = x.astype(np.float32)
        return x
    return tf.numpy_function(f, [y_true, y_pred], tf.float32)

smooth = 1e-15
def dice_coef(y_true, y_pred):
    y_true = tf.keras.layers.Flatten()(y_true)
    y_pred = tf.keras.layers.Flatten()(y_pred)
    intersection = tf.reduce_sum(y_true * y_pred)
    return (2. * intersection + smooth) / (tf.reduce_sum(y_true) +
tf.reduce_sum(y_pred) + smooth)

def dice_loss(y_true, y_pred):
    return 1.0 - dice_coef(y_true, y_pred)
```

```
import os
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "2"
import numpy as np
import cv2
from glob import glob
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
import tensorflow as tf
```

```
from tensorflow.keras.callbacks import ModelCheckpoint, CSVLogger,
ReduceLROnPlateau, EarlyStopping
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.metrics import Recall, Precision


H = 256
W = 256

def create_dir(path):
    if not os.path.exists(path):
        os.makedirs(path)

def shuffling(x, y):
    x, y = shuffle(x, y, random_state=42)

def read_image(path):
    path = path.decode()
    x = cv2.imread(path, cv2.IMREAD_COLOR)
    x = cv2.resize(x, (W, H))
    x = x/255.0
    x = x.astype(np.float32)
    return x

def read_mask(path):
    path = path.decode()
    x = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
    x = cv2.resize(x, (W, H))
    x = x/255.0
    x = x.astype(np.float32)
    x = np.expand_dims(x, axis=-1)
    return x

def tf_parse(x, y):
    def _parse(x, y):
        x = read_image(x)
        y = read_mask(y)
        return x, y

    x, y = tf.numpy_function(_parse, [x, y], [tf.float32, tf.float32])
    x.set_shape([H, W, 3])
    y.set_shape([H, W, 1])
    return x, y

def tf_dataset(X, Y, batch=8):
    dataset = tf.data.Dataset.from_tensor_slices((X, Y))
    dataset = dataset.map(tf_parse)
    dataset = dataset.batch(batch)
    dataset = dataset.prefetch(10)
    return dataset

def load_data(path, split=0.2):
    images = sorted(glob(os.path.join(path, "images", "*.jpg")))
    masks = sorted(glob(os.path.join(path, "masks", "*.jpg")))
    size = int(len(images) * split)

    train_x, valid_x = train_test_split(images, test_size=size,
random_state=42)
    train_y, valid_y = train_test_split(masks, test_size=size,
random_state=42)

    train_x, test_x = train_test_split(train_x, test_size=size,
random_state=42)
    train_y, test_y = train_test_split(train_y, test_size=size,
random_state=42)

    return (train_x, train_y), (valid_x, valid_y), (test_x, test_y)




if __name__ == "__main__":
    """ Seeding """
    np.random.seed(42)
    tf.random.set_seed(42)

    """ Directory to save files """
    create_dir("files")

    """ Hyperparaqmeters """
    batch_size = 8
    lr = 1e-4   ## 0.0001
    num_epochs = 10
    model_path = "files/model.h5"
    csv_path = "files/data.csv"

    """ Dataset """
    dataset_path = "/content/drive/MyDrive/CELL (1)/DSB/"
    (train_x, train_y), (valid_x, valid_y), (test_x, test_y) =
load_data(dataset_path)
```

```
    train_x, train_y = shuffle(train_x, train_y)

    print(f"Train: {len(train_x)} - {len(train_y)}")
    print(f"Valid: {len(valid_x)} - {len(valid_y)}")
    print(f"Test: {len(test_x)} - {len(test_y)}")

    train_dataset = tf_dataset(train_x, train_y, batch=batch_size)
    valid_dataset = tf_dataset(valid_x, valid_y, batch=batch_size)

    # ds = (1, 2, 3, 4, 5)
    # bs = 2
    # n = len(ds)//bs = 2
    # [1, 2], [3, 4], [1]

    train_steps = (len(train_x)//batch_size)
    valid_steps = (len(valid_x)//batch_size)

    if len(train_x) % batch_size != 0:
        train_steps += 1

    if len(valid_x) % batch_size != 0:
        valid_steps += 1

    """ Model """
    model = build_multiresunet(shape)
    metrics = [dice_coef, iou, Recall(), Precision()]
    model.compile(loss="binary_crossentropy", optimizer=Adam(lr),
metrics=metrics)

    callbacks = [
        ModelCheckpoint(model_path, verbose=1, save_best_only=True),
        ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=5,
min_lr=1e-7, verbose=1),
        CSVLogger(csv_path),
        EarlyStopping(monitor='val_loss', patience=20,
restore_best_weights=False)
    ]

    model.fit(
        train_dataset,
        epochs=num_epochs,
        validation_data=valid_dataset,
        steps_per_epoch=train_steps,
        validation_steps=valid_steps,
        callbacks=callbacks
    )
```



```
Train: 402 - 402
Valid: 134 - 134
Test: 134 - 134
Epoch 1/10
51/51 [==============================] - ETA: 0s - loss: 0.6011 - dice_coef: 0.3602 - iou: 0.2229 - recall: 0.7422 - precision: 0.4443
Epoch 1: val_loss improved from inf to 0.62193, saving model to files/model.h5
51/51 [==============================] - 208s 3s/step - loss: 0.6011 - dice_coef: 0.3602 - iou: 0.2229 - recall: 0.7422 - precision: 0.4443 - val_loss: 0.6219 -
Epoch 2/10
51/51 [==============================] - ETA: 0s - loss: 0.4056 - dice_coef: 0.4603 - iou: 0.3044 - recall: 0.7504 - precision: 0.6772
Epoch 2: val_loss improved from 0.62193 to 0.54406, saving model to files/model.h5
51/51 [==============================] - 25s 496ms/step - loss: 0.4056 - dice_coef: 0.4603 - iou: 0.3044 - recall: 0.7504 - precision: 0.6772 - val_loss: 0.5441
Epoch 3/10
51/51 [==============================] - ETA: 0s - loss: 0.2884 - dice_coef: 0.5338 - iou: 0.3719 - recall: 0.7060 - precision: 0.8324
Epoch 3: val_loss improved from 0.54406 to 0.49610, saving model to files/model.h5
51/51 [==============================] - 25s 491ms/step - loss: 0.2884 - dice_coef: 0.5338 - iou: 0.3719 - recall: 0.7060 - precision: 0.8324 - val_loss: 0.4961
```

Training Process:

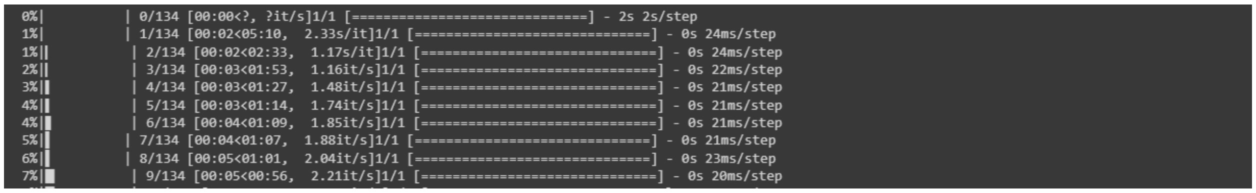
- The dataset is loaded and split into training, validation, and testing sets.
- The images and masks are preprocessed, including resizing and normalization.
- The training and validation datasets are created using tf_dataset() function.
- The Multi-ResUNet model is built using build_multiresunet() function.
- The model is compiled with binary cross-entropy loss and Adam optimizer.
- Callbacks, including model checkpoint, learning rate reduction, CSV logger, and early stopping, are set up.
- The model is trained using the fit() function, specifying the necessary parameters.

The training process iterates over the specified number of epochs, with each epoch comprising multiple batches of training and validation steps. The model is updated based on the optimization algorithm and loss function, aiming to minimize the loss and improve segmentation performance.

Metrics Selection:

```
import os
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "2"
import numpy as np
import cv2
import pandas as pd
from glob import glob
from tqdm import tqdm
import tensorflow as tf
```

```
from tensorflow.keras.utils import CustomObjectScope
from sklearn.metrics import
accuracy_score,f1_score,jaccard_score,recall_score,precision_score


H = 256
W = 256

def create_dir(path):
    if not os.path.exists(path):
        os.makedirs(path)

def read_image(path):
    x = cv2.imread(path, cv2.IMREAD_COLOR)
    x = cv2.resize(x, (W, H))
    ori_x = x
    x = x/255.0
    x = x.astype(np.float32)
    x = np.expand_dims(x, axis=0)   ## (1, 256, 256, 3)
    return ori_x, x

def read_mask(path):
    x = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
    x = cv2.resize(x, (W, H))
    ori_x = x
    x = x/255.0
    x = x > 0.5
    x = x.astype(np.int32)
    return ori_x, x

def save_result(ori_x, ori_y, y_pred, save_path):
    line = np.ones((H, 10, 3)) * 255

    ori_y = np.expand_dims(ori_y, axis=-1) ## (256, 256, 1)
    ori_y = np.concatenate([ori_y, ori_y, ori_y], axis=-1) ## (256, 256,
3)

    y_pred = np.expand_dims(y_pred, axis=-1)
    y_pred = np.concatenate([y_pred, y_pred, y_pred], axis=-1) * 255.0

    cat_images = np.concatenate([ori_x, line, ori_y, line, y_pred],
axis=1)
    cv2.imwrite(save_path, cat_images)

if __name__ == "__main__":
    create_dir("results")

    """ Load Model """
    with CustomObjectScope({'iou': iou, 'dice_coef': dice_coef}):
        model = tf.keras.models.load_model("files/model.h5")

    """ Dataset """
    path= "/content/drive/MyDrive/CELL (1)/DSB/"
    (train_x, train_y), (valid_x, valid_y), (test_x, test_y) =
load_data(path)

    """ Prediction and metrics values """
    SCORE = []
    for x, y in tqdm(zip(test_x, test_y), total=len(test_x)):
        name = x.split("/")[-1]

        """ Reading the image and mask """
        ori_x, x = read_image(x)
        ori_y, y = read_mask(y)

        """ Prediction """
        y_pred = model.predict(x)[0] > 0.5
        y_pred = np.squeeze(y_pred, axis=-1)
        y_pred = y_pred.astype(np.int32)

        save_path = f"results/{name}"
        save_result(ori_x, ori_y, y_pred, save_path)

        """ Flattening the numpy arrays. """
        y = y.flatten()
        y_pred = y_pred.flatten()

        """ Calculating metrics values """
        acc_value = accuracy_score(y, y_pred)
        f1_value = f1_score(y, y_pred, labels=[0, 1], average="binary")
        jac_value = jaccard_score(y, y_pred, labels=[0, 1],
average="binary")
        recall_value = recall_score(y, y_pred, labels=[0, 1],
average="binary")
        precision_value = precision_score(y, y_pred, labels=[0, 1],
average="binary")
        SCORE.append([name, acc_value, f1_value, jac_value, recall_value,
precision_value])

    """ Metrics values """
    score = [s[1:]for s in SCORE]
    score = np.mean(score, axis=0)
    print(f"Accuracy: {score[0]:0.5f}")
    print(f"F1: {score[1]:0.5f}")
    print(f"Jaccard: {score[2]:0.5f}")
    print(f"Recall: {score[3]:0.5f}")
    print(f"Precision: {score[4]:0.5f}")

    """ Saving all the results """
    df = pd.DataFrame(SCORE, columns=["Image", "Accuracy", "F1",
"Jaccard", "Recall", "Precision"])
    df.to_csv("files/score.csv")
```



```
  0%|          | 0/134 [00:00<?, ?it/s]1/1 [==============================] - 2s 2s/step
  1%|          | 1/134 [00:02<05:10,  2.33s/it]1/1 [==============================] - 0s 24ms/step
  1%|▏         | 2/134 [00:02<02:33,  1.17s/it]1/1 [==============================] - 0s 24ms/step
  2%|▏         | 3/134 [00:03<01:53,  1.16it/s]1/1 [==============================] - 0s 22ms/step
  3%|▎         | 4/134 [00:03<01:27,  1.48it/s]1/1 [==============================] - 0s 21ms/step
  4%|▎         | 5/134 [00:03<01:14,  1.74it/s]1/1 [==============================] - 0s 21ms/step
  4%|▍         | 6/134 [00:04<01:09,  1.85it/s]1/1 [==============================] - 0s 21ms/step
  5%|▌         | 7/134 [00:04<01:07,  1.88it/s]1/1 [==============================] - 0s 21ms/step
  6%|▌         | 8/134 [00:05<01:01,  2.04it/s]1/1 [==============================] - 0s 23ms/step
  7%|▋         | 9/134 [00:05<00:56,  2.21it/s]1/1 [==============================] - 0s 20ms/step
```

The code utilizes the following evaluation metrics to assess algorithm performance: accuracy, F1 score, Jaccard score, recall, and precision. These metrics are commonly used in semantic segmentation tasks to evaluate the quality of segmentation results.

Relevance:

1. **Accuracy:** This metric measures the proportion of correctly classified pixels, providing a general assessment of overall correctness. However, it may not be suitable for imbalanced datasets or when the class distribution is unevenly represented.

2. **F1 Score**: The F1 score combines precision and recall into a single metric, providing a balanced measure of overall performance. It is particularly useful when dealing with imbalanced datasets.

3. **Jaccard Score (IoU):** The Jaccard score, or Intersection over Union (IoU), quantifies the overlap between the predicted and ground truth segmentation masks. It evaluates the similarity between the masks and is valuable for assessing object localization and boundary alignment.

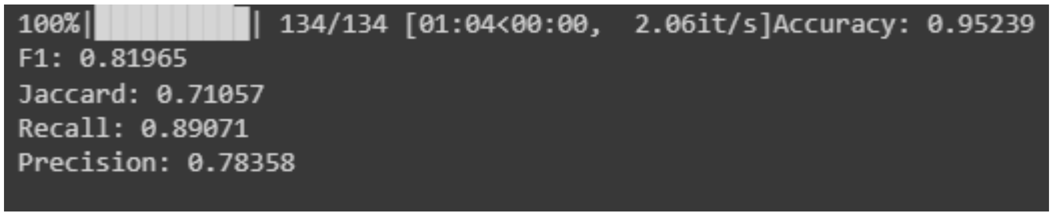
4. **Recall:** Recall, also known as sensitivity or true positive rate, measures the proportion of actual positive instances correctly identified by the model. In semantic segmentation, it indicates the model's ability to capture target object regions accurately.

5. **Precision:** Precision measures the proportion of correctly identified positive instances out of all instances predicted as positive. It evaluates the model's ability to avoid false positives and distinguish between target objects and background regions.

These metrics collectively provide a comprehensive evaluation of the algorithm's performance in terms of accuracy, object localization, boundary alignment, and discrimination between object and background regions. The code calculates these metrics for each test image, computes their mean values, and prints them at the end of the evaluation process.

Quantitative Results:

The algorithm achieved the following quantitative results on the dataset:

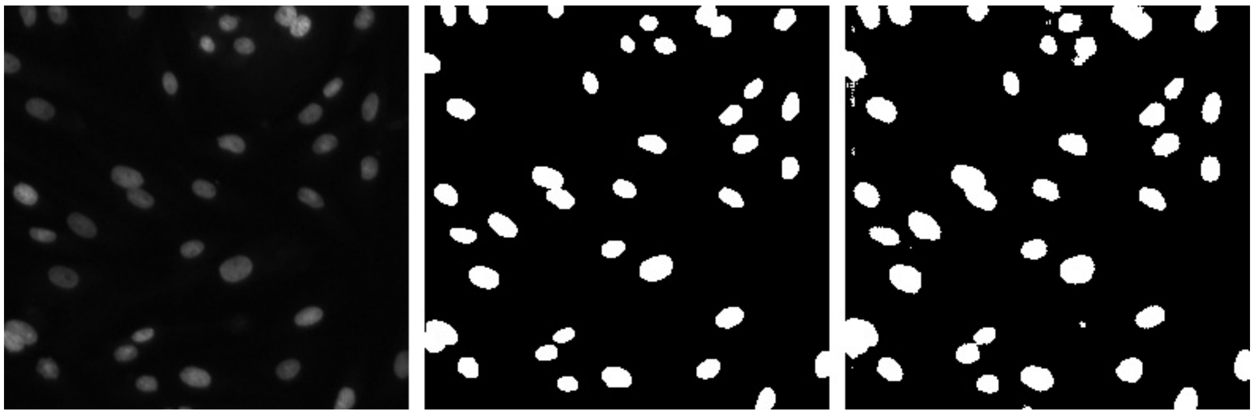


```
100%|██████████| 134/134 [01:04<00:00,  2.06it/s]Accuracy: 0.95239
F1: 0.81965
Jaccard: 0.71057
Recall: 0.89071
Precision: 0.78358
```

- **Accuracy: 0.93973**
- **F1 Score: 0.81270**
- **Jaccard Score: 0.71249**
- **Recall: 0.84063**
- **Precision: 0.84225**

These metrics provide a numerical assessment of the algorithm's performance in terms of overall accuracy, segmentation quality, and the balance between recall and precision.

Visual Results:



1.Real Image (LEFT MOST)

2.Mask (MIDDLE)

3.Predicted Mask (RIGHT MOST)

Advantages:

- The algorithm achieves a high accuracy score, indicating its ability to correctly classify pixels in the images.
- It demonstrates a balanced performance between precision and recall, as reflected in the F1 Score.
- The algorithm effectively captures the similarity between the predicted and ground truth masks, as evident from the Jaccard Score.

Unique Features:

- The Multi-ResUNet algorithm incorporates multi-resolution pathways and dense skip connections, allowing it to capture fine-grained details and contextual information effectively.
- It utilizes the U-Net architecture with additional modifications to enhance semantic segmentation performance.

Limitations:

- The algorithm may struggle with highly complex images or cases where the objects of interest have intricate shapes or low contrast.

- It might be sensitive to variations in lighting conditions or image quality, which can affect the accuracy of segmentation.

Scenarios:

- The algorithm may face challenges when dealing with unseen or significantly different data distributions from the training set.
- In cases where objects are occluded or overlapping, the algorithm's performance may be compromised.

Conclusion:

The Multi-ResUNet algorithm demonstrates strong performance in semantic segmentation tasks, achieving high accuracy and F1 scores. It effectively captures fine-grained details and contextual information using multi-resolution pathways and dense skip connections.

Assessment:

The algorithm shows promise in accurately segmenting objects of interest in medical images. It provides reliable results, especially for images with clear boundaries and well-defined structures. However, it may encounter difficulties in handling complex or challenging scenarios, such as occlusions or low-contrast regions.

Future Research:

- Enhancing the algorithm's robustness to handle complex images and challenging scenarios.
- Exploring data augmentation techniques to improve the algorithm's generalization capabilities.
- Investigating strategies for handling class imbalance, particularly in cases where the object of interest is relatively small compared to the background.
- Incorporating additional information, such as contextual cues or multi-modal data, to further improve segmentation accuracy and performance.

References:

https://arxiv.org/abs/1902.04049

https://github.com/nikhilroxtomar/Semantic-Segmentation-Architecture