INTERNSHIP
ARCHITECTURE -6
VGG19
REPORT

BY:

NAME: VANGA KANISHKA NADH

ROLL: 201CS165

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

MENOTR:

PROF. JENY RAJAN

Introduction:

VGG-19 is a deep convolutional neural network (CNN) architecture designed for image recognition tasks. It was developed by the Visual Geometry Group (VGG) at the University of Oxford and was one of the participants in the 2014 ImageNet Large Scale Visual Recognition Challenge (ILSVRC). The primary objective of VGG-19 is to classify images into various categories, such as objects, animals, and scenes, by learning hierarchical features from the input images.

Key Principles:

1. Depth: VGG-19 is characterized by its depth, which refers to the number of convolutional layers it employs. It consists of 19 layers, including 16 convolutional layers and 3 fully connected layers, making it deeper than previous CNN architectures like AlexNet.
2. Uniform Architecture: VGG-19 follows a uniform architecture, where each layer, except the last few, consists of a 3x3 convolutional kernel with a stride of 1 and a padding of 1, followed by a ReLU activation function. This design choice makes it easy to implement and understand.

3. Max Pooling: After a series of convolutional layers, VGG-19 uses max-pooling layers to reduce spatial dimensions, effectively capturing the most salient features while reducing computation.

4. Fully Connected Layers: Towards the end of the network, VGG-19 employs fully connected layers that help in making the final predictions based on the learned features.

Relevant Papers:

Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." arXiv preprint arXiv:1409.1556 (2014). (This paper introduces the VGG architecture and its variations.)

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 architecture used in this code snippet is a U-Net with a VGG-19 backbone. U-Net is a popular convolutional neural network architecture for image segmentation tasks, known for its encoder-decoder design. The encoder part captures features from the input image, and the decoder part upsamples the features to produce the final segmentation map.

Key Components:

1. Encoder (VGG-19 Backbone): The VGG-19 model is used as the encoder, responsible for extracting hierarchical features from the input image. It consists of multiple convolutional layers with max-pooling to reduce spatial dimensions and capture low to high-level features.

2. Decoder: The decoder part of the U-Net is implemented using the decoder_block function. This function takes the output from the previous layer and the corresponding feature maps from the encoder (skip connections) and performs upsampling using Conv2DTranspose layers. It then concatenates the upsampled features with the skip

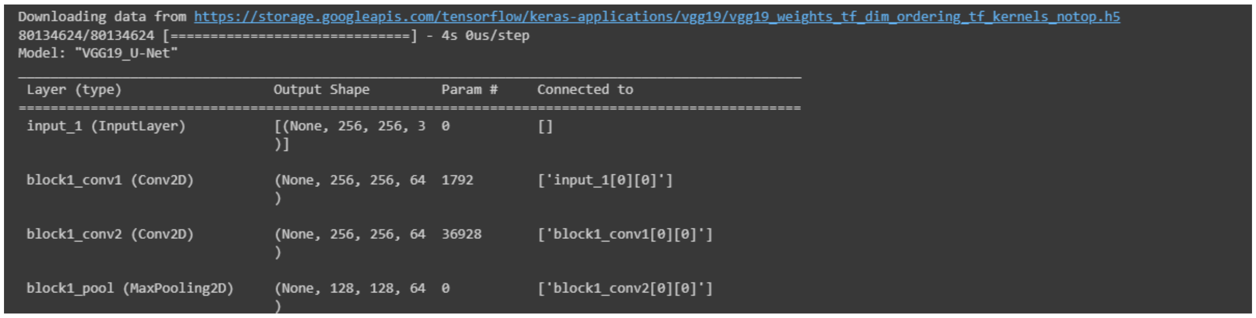
connections and passes the result through two convolutional blocks using the conv_block function.

3. Bridge: The bridge connects the encoder and the decoder. In this implementation, it uses the output of one of the convolutional blocks from the VGG-19 encoder as the bridge feature (b1), which is then passed to the decoder.
4. Output: The final output of the model is generated by a Conv2D layer with one filter and a sigmoid activation function, which produces a segmentation map of the same dimensions as the input image, representing the binary segmentation mask**.**

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

```
from tensorflow.keras.layers import Conv2D, BatchNormalization,
Activation, MaxPool2D, Conv2DTranspose, Concatenate, Input
from tensorflow.keras.models import Model
from tensorflow.keras.applications import VGG19

def conv_block(input, num_filters):
    x = Conv2D(num_filters, 3, padding="same")(input)
    x = BatchNormalization()(x)
    x = Activation("relu")(x)

    x = Conv2D(num_filters, 3, padding="same")(x)
    x = BatchNormalization()(x)
    x = Activation("relu")(x)

    return x

def decoder_block(input, skip_features, num_filters):
    x = Conv2DTranspose(num_filters, (2, 2), strides=2, padding="same")
(input)
    x = Concatenate()([x, skip_features])
    x = conv_block(x, num_filters)
    return x

def build_vgg19_unet(input_shape):
    """ Input """
    inputs = Input(input_shape)
    """ Pre-trained VGG19 Model """
    vgg19 = VGG19(include_top=False, weights="imagenet",
input_tensor=inputs)

    """ Encoder """
    s1 = vgg19.get_layer("block1_conv2").output         ## (512 x 512)
    s2 = vgg19.get_layer("block2_conv2").output         ## (256 x 256)
    s3 = vgg19.get_layer("block3_conv4").output         ## (128 x 128)
    s4 = vgg19.get_layer("block4_conv4").output         ## (64 x 64)

    """ Bridge """
    b1 = vgg19.get_layer("block5_conv4").output         ## (32 x 32)

    """ Decoder """
    d1 = decoder_block(b1, s4, 512)                     ## (64 x 64)
    d2 = decoder_block(d1, s3, 256)                     ## (128 x 128)
    d3 = decoder_block(d2, s2, 128)                     ## (256 x 256)
    d4 = decoder_block(d3, s1, 64)                      ## (512 x 512)

    """ Output """
    outputs = Conv2D(1, 1, padding="same", activation="sigmoid")(d4)

    model = Model(inputs, outputs, name="VGG19_U-Net")
    return model

if __name__ == "__main__":
    input_shape = (256, 256, 3)
    model = build_vgg19_unet(input_shape)
    model.summary()
```

Output:



```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg19/vgg19_weights_tf_dim_ordering_tf_kernels_notop.h5
80134624/80134624 [==============================] - 4s 0us/step
Model: "VGG19_U-Net"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_1 (InputLayer)           [(None, 256, 256, 3  0           []
                                )]

 block1_conv1 (Conv2D)          (None, 256, 256, 64  1792        ['input_1[0][0]']
                                )

 block1_conv2 (Conv2D)          (None, 256, 256, 64  36928       ['block1_conv1[0][0]']
                                )

 block1_pool (MaxPooling2D)     (None, 128, 128, 64  0           ['block1_conv2[0][0]']
                                )
```

<u>Training Methodology and Parameters:</u>

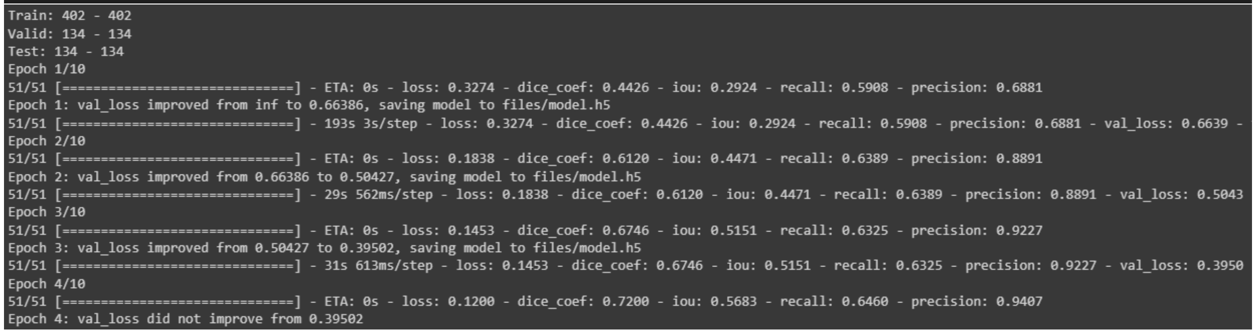
1. Optimization Algorithm: The optimization algorithm employed for training is Adam. The Adam optimizer adapts the learning rates of individual model parameters based on estimates of the first and second moments of the gradients. It is well-suited for training deep neural networks and helps in achieving faster convergence.
2. Loss Function: The loss function used during training is the "binary_crossentropy." This loss function is commonly used for binary image segmentation tasks, where the goal is to predict a binary mask (foreground vs. background) for each input image. Binary cross-entropy measures the dissimilarity between the predicted probabilities and the true binary labels.
3. Metrics: The model uses several metrics to monitor its performance during training:
    - Dice Coefficient: Measures the similarity between the predicted and true segmentation masks. A higher dice coefficient indicates better segmentation accuracy.
    - Intersection over Union (IoU): Also known as the Jaccard Index, it measures the overlap between the predicted and true masks. Higher IoU values represent better segmentation performance.
    - Recall: Measures the fraction of true positive instances among all positive instances. A higher recall indicates better sensitivity.
    - Precision: Measures the fraction of true positive instances among all predicted positive instances. A higher precision indicates fewer false positives.
4. Data Preprocessing: The code includes functions to read and preprocess images and masks. The images are loaded in color (RGB) format, resized to a fixed height and width (H=256, W=256), and normalized to have values between 0 and 1. The masks are loaded in grayscale format, resized, and also normalized to values between 0 and 1.
5. Dataset Splitting: The dataset is split into training, validation, and test sets. The split ratios for validation and test sets are specified by the split parameter (defaulted to 0.2).
6. Data Shuffling: The training data is shuffled using the shuffle function before being used for training. Shuffling helps randomize the order of data samples, reducing any inherent biases and improving the training process.

```
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)

    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_vgg19_unet((H, W, 3))
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: 0.3274 - dice_coef: 0.4426 - iou: 0.2924 - recall: 0.5908 - precision: 0.6881
Epoch 1: val_loss improved from inf to 0.66386, saving model to files/model.h5
51/51 [==============================] - 193s 3s/step - loss: 0.3274 - dice_coef: 0.4426 - iou: 0.2924 - recall: 0.5908 - precision: 0.6881 - val_loss: 0.6639 -
Epoch 2/10
51/51 [==============================] - ETA: 0s - loss: 0.1838 - dice_coef: 0.6120 - iou: 0.4471 - recall: 0.6389 - precision: 0.8891
Epoch 2: val_loss improved from 0.66386 to 0.50427, saving model to files/model.h5
51/51 [==============================] - 29s 562ms/step - loss: 0.1838 - dice_coef: 0.6120 - iou: 0.4471 - recall: 0.6389 - precision: 0.8891 - val_loss: 0.5043
Epoch 3/10
51/51 [==============================] - ETA: 0s - loss: 0.1453 - dice_coef: 0.6746 - iou: 0.5151 - recall: 0.6325 - precision: 0.9227
Epoch 3: val_loss improved from 0.50427 to 0.39502, saving model to files/model.h5
51/51 [==============================] - 31s 613ms/step - loss: 0.1453 - dice_coef: 0.6746 - iou: 0.5151 - recall: 0.6325 - precision: 0.9227 - val_loss: 0.3950
Epoch 4/10
51/51 [==============================] - ETA: 0s - loss: 0.1200 - dice_coef: 0.7200 - iou: 0.5683 - recall: 0.6460 - precision: 0.9407
Epoch 4: val_loss did not improve from 0.39502
```

Training Process:

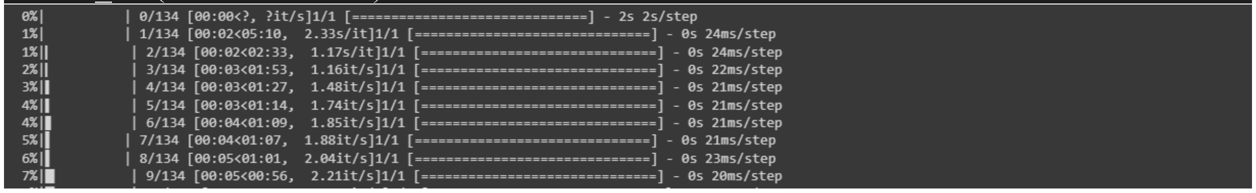
The model is trained using the **fit** function. The training data is loaded as a TensorFlow dataset using the **tf_dataset** function, and the validation data is also loaded as a separate dataset. The model is compiled with the Adam optimizer, binary cross-entropy loss, and the specified metrics. Training is performed for a specified number of epochs (defaulted to 10), and during each epoch, the model's performance on the validation set is monitored.

Several callbacks are used during training to monitor and control the training process effectively:

- ModelCheckpoint: Saves the best model based on the validation loss to the specified **model_path**.
- ReduceLROnPlateau: Reduces the learning rate by a factor of 0.1 if the validation loss plateaus, which helps fine-tune the model's performance.
- CSVLogger: Logs the training and validation metrics to a CSV file specified by **csv_path**.
- EarlyStopping: Stops the training process early if the validation loss does not improve for a certain number of epochs (patience).

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.

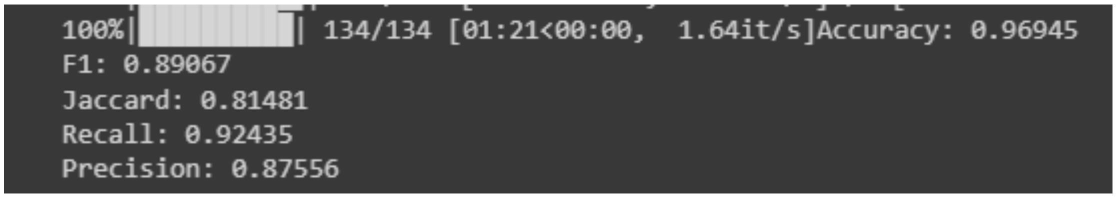
<u>**Relevance:**</u>

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 [01:21<00:00,  1.64it/s]Accuracy: 0.96945
F1: 0.89067
Jaccard: 0.81481
Recall: 0.92435
Precision: 0.87556
```

Accuracy: 0.96945

F1: 0.89067

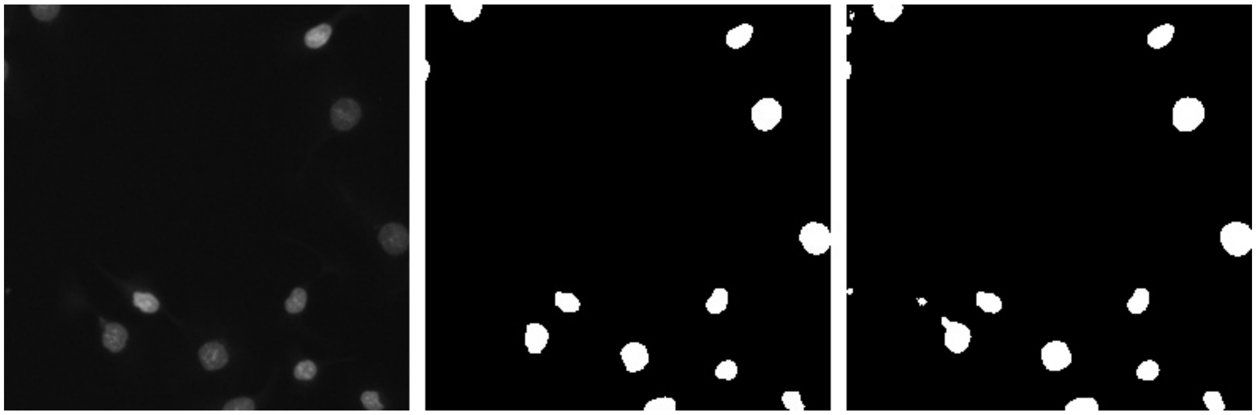
Jaccard: 0.81481

Recall: 0.92435

Precision: 0.87556

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.

<u>Visual Results:</u>



1.Real Image (LEFT MOST)

2.Mask (MIDDLE)

3.Predicted Mask (RIGHT MOST)

<u>Advantages:</u>

1. Hierarchical Feature Learning: VGG-19 effectively captures hierarchical features from images, enabling better understanding of complex visual patterns.
2. Transfer Learning: Pre-trained VGG-19 models on ImageNet facilitate transfer learning, making it applicable to various image recognition tasks.
3. Versatility: VGG-19's uniform architecture allows easy implementation and modification, making it suitable for different computer vision tasks.

<u>Unique Features:</u>

1. Depth: VGG-19's 19-layer architecture enables learning high-level abstract features, contributing to its exceptional performance.
2. Wide Applicability: VGG-19 excels in image classification, object detection, and image segmentation tasks due to its hierarchical learning and transfer learning capabilities.

Limitations:

1. Computational Cost: VGG-19's large number of parameters leads to high computational requirements, limiting its use in real-time or resource-limited scenarios.
2. Overfitting: When training on small datasets, VGG-19 may overfit, necessitating regularization techniques and data augmentation.

Scenarios:

1. Real-time Applications: VGG-19 may not be suitable for real-time tasks due to its computational demands.
2. Limited Data: Overfitting concerns arise with small datasets; other architectures or transfer learning approaches might be more suitable.

Conclusion:

VGG-19 is a powerful algorithm for image recognition, leveraging hierarchical feature learning and transfer learning. Its wide applicability and transfer learning potential make it a valuable tool. However, its high computational cost and proneness to overfitting should be considered for specific use cases.

Assessment:

VGG-19 demonstrates excellent performance but may require careful handling in resource-constrained scenarios. It remains a fundamental architecture for computer vision research and applications.

Future Research:

1. Architecture Optimization: Research can focus on reducing VGG-19's computational complexity while maintaining performance.
2. Domain-Specific Fine-Tuning: Further study can improve fine-tuning on specific domains with limited data.
3. Hybrid Architectures: Exploring hybrid models with VGG-19 and attention mechanisms could enhance feature extraction.
4. Real-Time Adaptation: Research methods to adapt VGG-19 for real-time and resource-constrained environments.

References:

https://arxiv.org/abs/1409.1556

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