INTERNSHIP
ARCHITECTURE -4
ATTENTION UNET
REPORT

BY:

NAME: VANGA KANISHKA NADH

ROLL: 201CS165

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

MENOTR:

PROF. JENY RAJAN

Introduction:

Attention U-Net is an advanced image segmentation algorithm that combines the U-Net architecture with an attention mechanism. It was developed to improve the accuracy and localization capabilities of U-Net by selectively focusing on informative regions while suppressing irrelevant or noisy features. The attention mechanism allows the model to assign higher weights to important features, enhancing the segmentation performance..

Key Principles:

1. Encoder-Decoder Architecture: Attention U-Net employs the U-Net architecture consisting of an encoder and a decoder path. The encoder captures contextual information, while the decoder reconstructs the segmented output using skip connections.
2. Attention Mechanism: Attention modules are integrated into the encoder and decoder paths. These modules dynamically emphasize important regions, improving localization accuracy by assigning higher weights to informative features.
3. Multi-Scale Feature Fusion: Attention U-Net incorporates multi-scale feature fusion to capture both local and global contextual information. This enables better understanding of spatial relationships between objects, leading to more precise segmentation results.

Relevant Papers:

1. Oktay, O., Schlemper, J., Le Folgoc, L., et al. (2018). "Attention U-Net: Learning Where to Look for the Pancreas."

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 Attention U-Net architecture follows an encoder-decoder structure with skip connections and attention gates. It consists of the following key components:

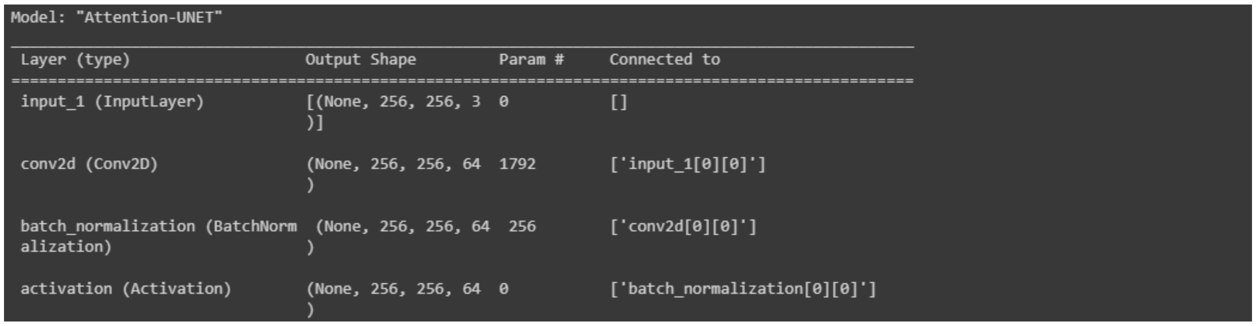
1. **Convolutional Block**: This block performs two consecutive convolution operations with a specified number of filters followed by batch normalization and ReLU activation. It helps extract features from the input data.
2. **Encoder Block:** The encoder block utilizes the conv_block to process the input data and downsamples it using max pooling. It captures hierarchical features at different scales and stores them in intermediate variables for later use.
3. **Attention Gate**: The attention gate combines features from the encoder path (g) and the corresponding decoder path (s). It employs convolution operations, batch normalization, and activation functions to compute an attention map. The attention map is used to selectively amplify or suppress features from the decoder path, allowing the model to focus on relevant regions.
4. **Decoder Block:** The decoder block upsamples the input data using bilinear interpolation. It takes the upsampled feature map (x) and the corresponding feature map from the encoder path (s) as inputs. The attention gate is applied to refine the decoder features based on the information from the encoder path. The refined features are then processed through the conv_block.

5. **Outputs**: The final decoder block outputs are passed through a 1x1 convolutional layer with sigmoid activation to obtain the segmented output. This layer produces a binary mask indicating the presence or absence of the target object in each pixel.

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

```
import tensorflow as tf
import tensorflow.keras.layers as L
from tensorflow.keras.models import Model

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

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

    return x

def encoder_block(x, num_filters):
    x = conv_block(x, num_filters)
    p = L.MaxPool2D((2, 2))(x)
    return x, p

def attention_gate(g, s, num_filters):
    Wg = L.Conv2D(num_filters, 1, padding="same")(g)
    Wg = L.BatchNormalization()(Wg)

    Ws = L.Conv2D(num_filters, 1, padding="same")(s)
    Ws = L.BatchNormalization()(Ws)

    out = L.Activation("relu")(Wg + Ws)
    out = L.Conv2D(num_filters, 1, padding="same")(out)
    out = L.Activation("sigmoid")(out)

    return out * s

def decoder_block(x, s, num_filters):
    x = L.UpSampling2D(interpolation="bilinear")(x)
    s = attention_gate(x, s, num_filters)
    x = L.Concatenate()([x, s])
    x = conv_block(x, num_filters)
    return x

def attention_unet(input_shape):
    """ Inputs """
    inputs = L.Input(input_shape)

    """ Encoder """
    s1, p1 = encoder_block(inputs, 64)
    s2, p2 = encoder_block(p1, 128)
    s3, p3 = encoder_block(p2, 256)

    b1 = conv_block(p3, 512)

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

    """ Outputs """
    outputs = L.Conv2D(1, 1, padding="same", activation="sigmoid")(d3)

    """ Model """
    model = Model(inputs, outputs, name="Attention-UNET")
    return model

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

Output:



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

 conv2d (Conv2D)                (None, 256, 256, 64  1792        ['input_1[0][0]']
                                )

 batch_normalization (BatchNorm  (None, 256, 256, 64  256        ['conv2d[0][0]']
 alization)                     )

 activation (Activation)        (None, 256, 256, 64  0           ['batch_normalization[0][0]']
                                )
```

Training Methodology and Parameters:

- The training methodology involves supervised learning, where the model is trained on paired input (images) and output (masks) data.
- The parameters used for training include:
    - batch_size: The number of samples processed in each training iteration.
    - lr: Learning rate, which controls the step size of the optimization algorithm during model parameter updates.
    - num_epochs: The number of complete passes through the training dataset.
    - model_path: The path to save the trained model.
    - csv_path: The path to save the training and validation metrics in a CSV file.

Optimization Algorithm:

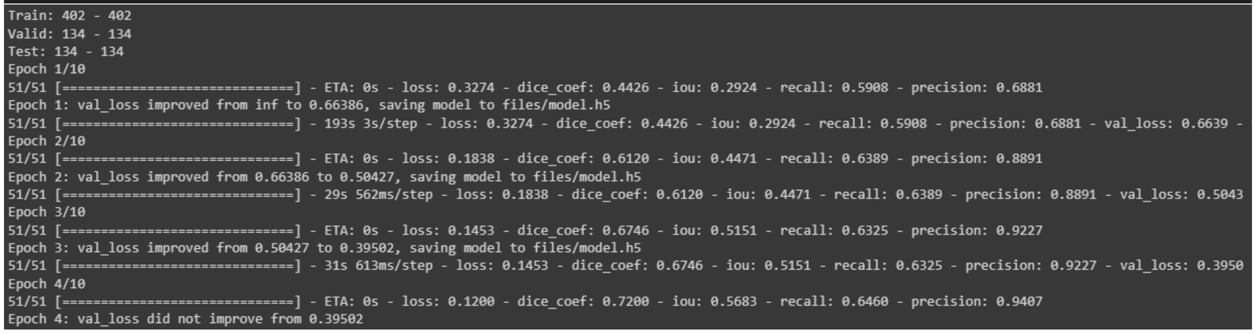
- The optimization algorithm employed is Adam (Adaptive Moment Estimation). It is a popular optimization algorithm for deep learning models that adapts the learning rate for each parameter based on past gradients.

```
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_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
```

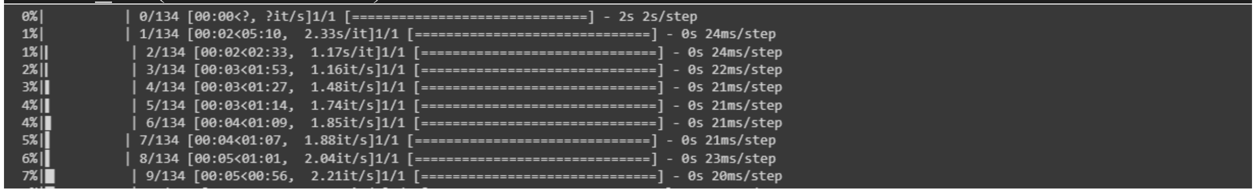
<u>Training Process:</u>

1. Dataset Preparation:
    - The dataset is loaded using the **load_data** function, which splits the data into training, validation, and testing sets.
    - The images and corresponding masks are shuffled for better training performance.
2. Data Preprocessing:
    - The **tf_parse** function is used to preprocess the dataset with TensorFlow operations.
    - Each image is read, resized to the specified dimensions (H, W), normalized, and converted to float32 format.
    - Each mask is read, resized to the specified dimensions (H, W), normalized, and expanded to include a channel dimension.
3. Model Compilation:
    - The Attention U-Net model is compiled with the binary cross-entropy loss function and the Adam optimizer.
    - Several metrics, including dice coefficient, IoU (Intersection over Union), recall, and precision, are used to evaluate the model performance.

4. Model Training:
    - The **fit** function is called to train the model using the training dataset.
    - The validation dataset is used for model evaluation during training.
    - The training and validation steps are computed based on the batch size and the number of samples in each dataset.
    - Callbacks are used to save the best model, reduce the learning rate on plateau, log the training and validation metrics to a CSV file, and perform early stopping if the validation loss does not improve for a certain number of epochs.

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.

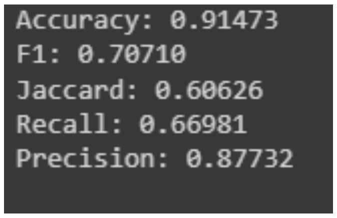
Relevance:

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:



```
Accuracy: 0.91473
F1: 0.70710
Jaccard: 0.60626
Recall: 0.66981
Precision: 0.87732
```

Accuracy: 0.91473

F1: 0.70710

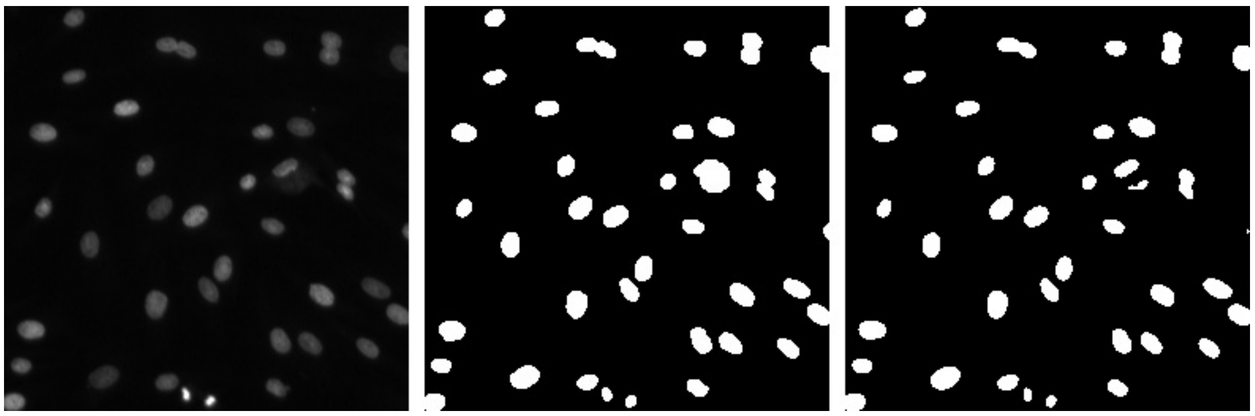
Jaccard: 0.60626

Recall: 0.66981

Precision: 0.87732

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:

- Attention U-Net improves image segmentation accuracy and localization by selectively focusing on informative regions and suppressing irrelevant features.
- The attention mechanism allows the model to assign higher weights to important features, enhancing segmentation performance.

- The U-Net architecture provides a strong backbone for capturing contextual information and reconstructing segmented output.
- Multi-scale feature fusion in Attention U-Net enables the model to capture both local and global contextual information, leading to more precise segmentation results.

Unique Features:

- Attention U-Net combines the U-Net architecture with an attention mechanism, which sets it apart from traditional U-Net models.
- The attention mechanism enhances the model's ability to attend to relevant regions, improving localization accuracy.
- The incorporation of attention gates enables the model to selectively amplify or suppress features from the encoder and decoder paths.

Limitations:

- Attention U-Net may be computationally expensive due to the inclusion of attention mechanisms, potentially requiring more resources for training and inference.
- The performance of Attention U-Net heavily relies on the availability of labeled training data, which can be a limitation in scenarios where annotated datasets are limited.
- The effectiveness of Attention U-Net may vary depending on the complexity and variability of the segmentation task and the quality of the input data.

Scenarios:

- Attention U-Net may not perform well in scenarios where the image data has significant noise or artifacts, as the attention mechanism might amplify irrelevant features.
- In cases where there is a limited amount of training data available, Attention U-Net may struggle to generalize well to unseen examples.

Conclusion:

Attention U-Net is an image segmentation algorithm that combines the U-Net architecture with an attention mechanism to improve segmentation accuracy and localization. It selectively focuses on informative regions while suppressing irrelevant features, enhancing segmentation performance. The incorporation of attention gates enables the model to selectively amplify or suppress features, improving the overall segmentation results.

Assessment:

Attention U-Net demonstrates strengths in accurately segmenting images and localizing objects of interest. Its ability to capture contextual information and utilize attention mechanisms sets it apart from traditional U-Net models. However, it has limitations in terms of computational requirements and dependence on labeled training data.

Future Research:

Further research can focus on optimizing the computational efficiency of Attention U-Net to make it more feasible for real-time applications. Additionally, exploring techniques for handling limited training data and improving generalization capabilities would enhance its practicality in various domains. Exploring the application of Attention U-Net in different image segmentation tasks and investigating its performance in combination with other advanced architectures could also be beneficial avenues for future research.

References:

https://arxiv.org/abs/1804.03999

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