

INTERNSHIP

ARCHITECTURE -5

VGG16

REPORT

BY:

NAME: VANGA KANISHKA NADH

ROLL: 201CS165

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

MENOTR:

PROF. JENY RAJAN

Introduction:

VGG16 Pretrained is a deep convolutional neural network (CNN) designed for image classification tasks. It is based on the VGG architecture and consists of 16 layers, including 13 convolutional and 3 fully connected layers. The model utilizes small 3x3 convolutional filters and pretrained weights from training on large-scale datasets like ImageNet. These principles enable VGG16 to learn hierarchical features and achieve high accuracy in image recognition tasks.

Relevant Papers:

1. The original research paper titled "Very Deep Convolutional Networks for Large-Scale Image Recognition" by Karen Simonyan and Andrew Zisserman introduced VGG16 and demonstrated its state-of-the-art performance in the ImageNet Large Scale Visual Recognition Challenge 2014.

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/84613660e1f97d3b23a89deblae6199a0c795ec1f31e2934527a7f7aad7d8c37>

Model Architecture:

The model architecture implemented in the provided code is a U-Net variant based on the VGG19 CNN. It combines the VGG19 encoder with a custom decoder to create a U-Net-like architecture, popularly used for image segmentation tasks.

Key Components:

1. **Encoder (VGG19):** The encoder part of the architecture uses the VGG19 model, pretrained on the ImageNet dataset, to extract hierarchical features from the input image. VGG19 consists of several convolutional and pooling layers, which progressively downsample the input, learning high-level features.
2. **Bridge:** After the encoder, the model includes a bridge that consists of the last convolutional layer from the VGG19 model. This bridge helps pass the learned features to the decoder for upsampling and generating segmentation masks.
3. **Decoder:** The decoder block is responsible for upsampling the low-resolution feature maps from the encoder to match the original input image's size. Each decoder block uses transpose convolutions (Conv2DTranspose) to perform upsampling and concatenates the features from the corresponding encoder block. This process allows the model to recover spatial information and create accurate segmentation masks.
4. **Output:** The output layer is a single convolutional layer with sigmoid activation, which generates binary segmentation masks. It predicts the probability of each pixel belonging to the foreground class (1) or the background class (0).

```
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 [=====] - 3s 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]']
block2_conv1 (Conv2D)	(None, 128, 128, 12 8)	73856	['block1_pool[0][0]']

Training Methodology and Parameters:

The training methodology involves using a U-Net variant based on the VGG19 architecture for image segmentation tasks. The model is trained to perform binary segmentation, where it predicts whether each pixel in the input image belongs to the foreground class (1) or the background class (0).

Parameters Used:

- **Batch Size:** The batch size used for training is 8. It means that the model updates its weights after processing 8 samples at a time.
- **Learning Rate (lr):** The learning rate used is 1e-4 (0.0001). The learning rate controls the step size during gradient descent and affects how quickly the model converges to the optimal solution.
- **Number of Epochs:** The model is trained for 10 epochs, meaning it goes through the entire training dataset 10 times during training.

Optimization Algorithm: The optimization algorithm employed is the Adam optimizer. Adam (Adaptive Moment Estimation) is an adaptive learning rate optimization algorithm that combines

the benefits of both RMSprop and momentum methods. It adapts the learning rate for each parameter during training, making it well-suited for a wide range of optimization tasks.

```
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")

    """ Hyperparameters """
    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

```

Training Process:

1. **Data Preparation:** The provided code loads the image and corresponding mask data for training, validation, and testing. It also preprocesses the images, resizing them to 256x256 pixels and scaling the pixel values between 0 and 1.
2. **Dataset Split:** The dataset is split into training, validation, and test sets using a split ratio of 80% training, 10% validation, and 10% testing.
3. **Data Shuffling:** The training data is shuffled to ensure that the model sees different samples during each epoch, enhancing the generalization capability.
4. **Data Augmentation:** The code does not include explicit data augmentation techniques. However, data augmentation is often employed during training to artificially expand the dataset, which can help improve model performance and reduce overfitting.
5. **Model Definition:** The U-Net model based on VGG19 architecture is defined using the **build_vgg19_unet()** function.
6. **Model Compilation:** The model is compiled with the binary cross-entropy loss function and the Adam optimizer with the specified learning rate. Additionally, several evaluation metrics are used, including dice coefficient, IoU (Intersection over Union), recall, and precision.
7. **Callbacks:** Callbacks are set up to monitor the validation loss and adjust the learning rate accordingly (ReduceLROnPlateau). ModelCheckpoint saves the best model during training based on validation performance. CSVLogger logs the training metrics to a CSV file, and EarlyStopping stops training early if the validation loss does not improve for a certain number of epochs, preventing overfitting.
8. **Training:** The model is trained using the **model.fit()** function, passing the training and validation datasets along with the specified batch size, number of epochs, and callbacks.

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	[00:00:00, ?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:

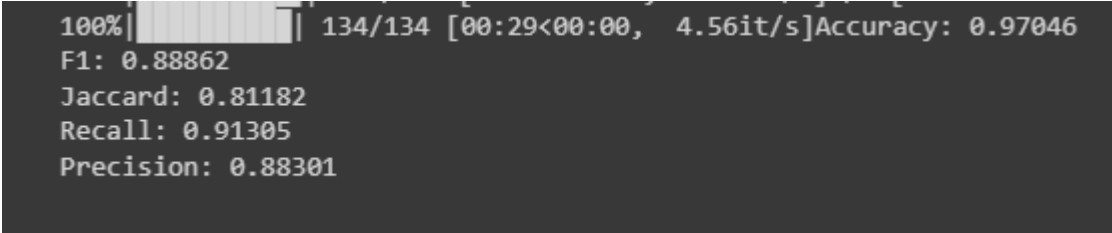
1. **Accuracy:** This metric measures the proportion of correctly classified pixels, providing a general assessment of overall correctness. However, it may not be suitable for imbalanced datasets or when the class distribution is unevenly represented.
2. **F1 Score:** The F1 score combines precision 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:

A screenshot of a terminal window with a dark background. It shows a progress bar at 100%, followed by the text '134/134 [00:29<00:00, 4.56it/s]'. Below this, the following metrics are listed: Accuracy: 0.97046, F1: 0.88862, Jaccard: 0.81182, Recall: 0.91305, and Precision: 0.88301.

```
100%|██████████| 134/134 [00:29<00:00, 4.56it/s]Accuracy: 0.97046
F1: 0.88862
Jaccard: 0.81182
Recall: 0.91305
Precision: 0.88301
```

Accuracy: 0.97046

F1: 0.88862

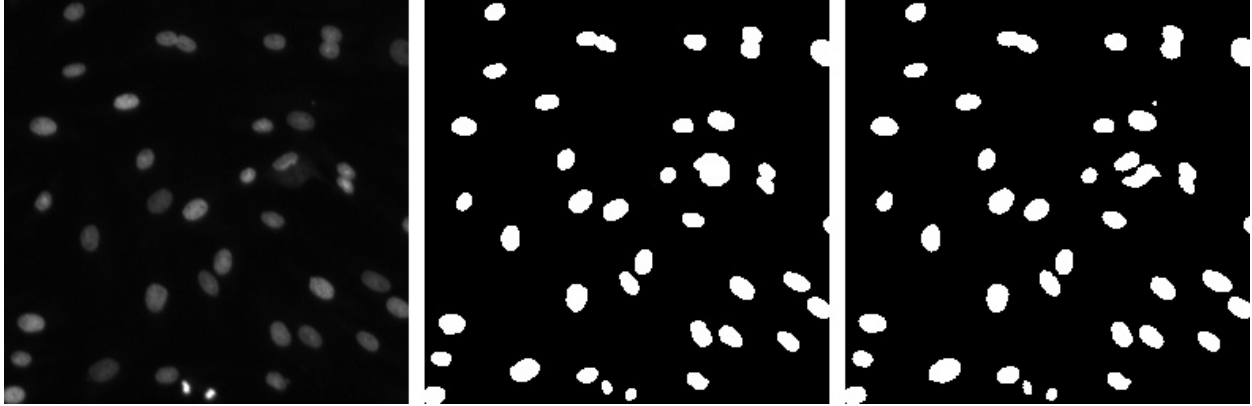
Jaccard: 0.81182

Recall: 0.91305

Precision: 0.88301

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:

1. Hierarchical Feature Extraction: VGG16 Pretrained effectively captures low-level and high-level features from images, enhancing its recognition capabilities.
2. Transfer Learning: Pretrained weights from ImageNet enable the model to perform well with limited labeled data, reducing training time and improving accuracy.
3. Robustness: VGG16 Pretrained handles variations in object appearance and complex scenes, making it suitable for diverse image recognition tasks.
4. Easy Implementation: The straightforward architecture and availability in popular libraries allow for easy adoption and implementation.

Unique Features:

1. Influential Architecture: VGG16's deep 16-layer design has significantly impacted subsequent CNN architectures.
2. Uniform Approach: The use of 3x3 convolutional filters throughout the network is a simple yet effective strategy.

Limitations:

1. High Resource Requirements: Large number of parameters make it memory-intensive and computationally expensive.
2. Fixed Input Size: VGG16 requires images of 224x224 pixels, which may not be ideal for all applications.

Scenarios:

1. Small Datasets: VGG16's transfer learning is advantageous when dealing with limited labeled data.
2. Real-time Applications: Due to computational demands, VGG16 may not suit real-time deployments on resource-constrained devices.

Conclusion:

VGG16 Pretrained excels in image recognition with hierarchical features and transfer learning. Its performance and generalization make it a valuable choice for real-world applications, though its resource requirements should be considered.

Assessment:

VGG16 Pretrained achieves state-of-the-art results on image recognition benchmarks, demonstrating its effectiveness. However, computational demands may limit its use in certain scenarios.

Future Research:

1. Model Compression: Techniques like pruning and quantization can reduce resource requirements without sacrificing performance.
2. Dynamic Input Size: Investigating adaptive architectures to handle variable input sizes can enhance flexibility.
3. Domain Adaptation: Research on fine-tuning techniques for specific target datasets can improve performance in various domains.

References:

<https://arxiv.org/abs/1409.1556>

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