Question 1:

Part 1:
The formula to calculate the size of a dilated kernel is:
Dilated Kernel Size $= k + (k - 1) \times (d - 1)$

Part 2:
The count of trainable parameters in a dilated convolution layer matches that of a standard convolution layer when the dilation rate is set to 1.

Part 3:
The receptive field in each layer can be determined using the following formulas:

1. Initial receptive field:
$RF_0 = 1$

2. Receptive field for subsequent layers:
$RF_i = RF_{i-1} + (k - 1) \times d$

Therefore, we have:
A=13 ,B=11 , C=51 , D=63 , E=7

Part 4:
1. Receptive field after convolution:
Receptive_out $=$ Receptive_in $+$ (kernel_size $- 1$) $\times$ dilation_rate

2. Receptive field after max pooling:
Receptive_out $=$ Receptive_in $+$ (kernel_size $- 1$) $\times$ stride

So we have:
$1 + 2 \times (5 - 1) + 3 \times$ (pool_size $- 1$) $\times$ stride $= 107$
$9 + 6 \times$ stride $= 107$
$6 \times$ stride $= 98$
stride ≈ 16.33

Question 2:

Part 1:
Regular Convolution

1. Number of Parameters:
   - Each filter has dimensions $5 \times 5 \times 3$.
   - There are 64 filters.
   - Total parameters: $5 \times 5 \times 3 \times 64 = 4800$

2. Number of Multiplications:
   - Output size: $128 \times 128 \times 64$
   - Each output pixel is calculated using $5 \times 5 \times 3 = 75$ multiplications.
   - Total multiplications: $128 \times 128 \times 64 \times 75 = 78643200$

Depthwise Separable Convolution

1. Depthwise Convolution:
   - Each of the 3 input channels is convolved with its own 5×5 filter.
   - Total parameters: $5 \times 5 \times 3 = 75$

2. Pointwise Convolution:
   - Each of the 3 outputs from the depthwise step is convolved with 64 1×1 filters.
   - Total parameters: $1 \times 1 \times 3 \times 64 = 192$

3. Total Parameters:
   - Depthwise parameters: 75
   - Pointwise parameters: 192
   - Total: $75 + 192 = 267$

4. Number of Multiplications:
   - Depthwise Convolution:
     - Output size: $128 \times 128 \times 3$
     - Each output pixel requires $5 \times 5 = 25$ multiplications.
     - Total multiplications: $128 \times 128 \times 3 \times 25 = 1228800$

- Pointwise Convolution:
  - Output size: $128 \times 128 \times 64$
  - Each output pixel requires 3 multiplications.
  - Total multiplications: $128 \times 128 \times 64 \times 3 = 3145728$

- Total Multiplications:
  $-1228800 + 3145728 = 4374528$

Part 2:
Regular Convolutional Layer

Parameters Calculation:
- Each filter has $3 \times 3 \times 32 = 288$ parameters.
- With 32 filters: $288 \times 32 = 9216$ parameters.

Depthwise Separable Convolution:

1. Depthwise Convolution:
   - Each input channel has its own 3×3 filter.
   - Total parameters for depthwise: $3 \times 3 \times 32 = 288$

2. Pointwise Convolution:
   - Each output of the depthwise layer (32 channels) is convolved with 32 1×1 filters.
   - Total parameters for pointwise: $1 \times 1 \times 32 \times 32 = 1024$

- Total for depthwise separable convolution: $288 + 1024 = 1312$

The number of parameters in the depthwise separable convolution compared to the regular convolution is: $\frac{1312}{9216} \approx 0.142$

Question 3:
Part 1:
Correlation Method:
- Pros:
  - Simple and fast to compute.
  - Effective with consistent intensity levels and alignment.
- Cons:
  - Sensitive to lighting and contrast variations.
  - Can produce high scores in high-intensity regions regardless of pattern match.

Normalized Cross-Correlation:
- Pros:
  - Robust to changes in lighting and contrast.
  - Focuses on pattern matching by normalizing intensity variations.
- Cons:
  - More computationally intensive.
  - Slightly more complex to implement.

Which Method is Better for This Question?
Normalized Cross-Correlation (NCC) is generally the better choice because of following reasons:
Robustness: NCC is more robust to variations in brightness and contrast, which makes it a more reliable method for template matching, especially when the image conditions are not controlled or uniform, just like the given image.
Accuracy: By normalizing the values, NCC focuses on the shape and pattern within the template and image region, leading to more accurate matching results.

Part 2: the notebook file (Q3.ipynb) is attached and the codes have been explained via comments in notebook.
output:

Question 4:
the notebook file (Q4.ipynb) is attached, modifed codes:

Convert BGR to RGB

```
image_rgb =  cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)
```

- Initialize the mask annotator with a color lookup index for coloring the masks
- Convert SAM result to a Detections object suitable for further processing and visualization
- Annotate the original image with the generated masks, producing an annotated image

```
mask_annotator = sv.MaskAnnotator(color_lookup=sv.ColorLookup.INDEX)
detections = sv.Detections.from_sam(sam_result=sam_result)
annotated_image = mask_annotator.annotate(scene=image_bgr.copy(), detections=detections)
```

output:

segmented image

- Get the first bounding box from the widget's list of bounding boxes
- Convert the bounding box from the widget format (x, y, width, height) to the format required by the mask predictor (x_min, y_min, x_max, y_max)

```
box = widget.bboxes[0]

box = np.array([
    box['x'],
    box['y'],
    box['x'] + box['width'],
    box['y'] + box['height']
])
```

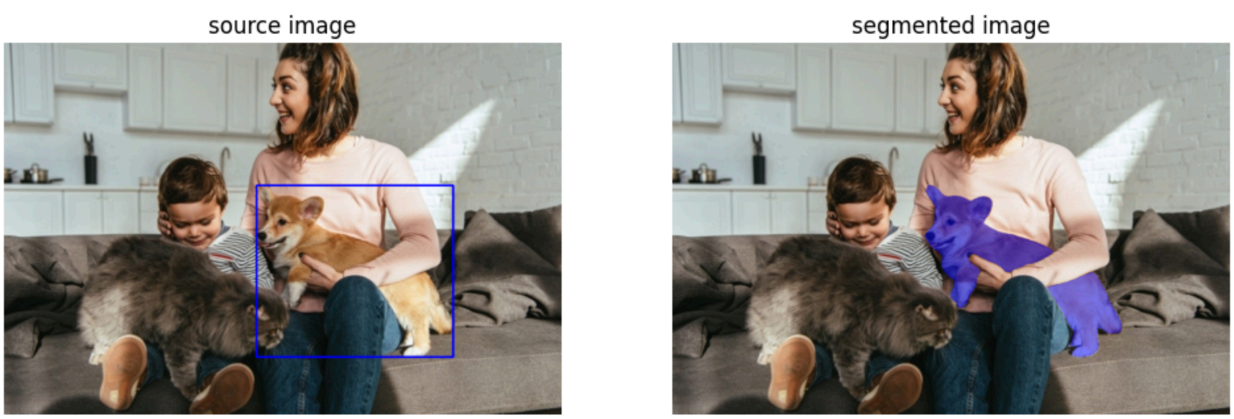
- Initialize the box annotator with a blue color for drawing bounding boxes
- Initialize the mask annotator with a blue color for drawing masks and use a color lookup index for coloring the masks

```
box_annotator = sv.BoxAnnotator(color=sv.Color.blue())


mask_annotator = sv.MaskAnnotator(color=sv.Color.blue(), color_lookup=sv.ColorLookup.INDEX)
```

- Annotate the original image with bounding boxes from the detections. Use a copy of the original image and skip adding labels to the bounding boxes.
- Annotate the original image with masks from the detections. Use a copy of the original image to overlay the masks.

```
source_image = box_annotator.annotate(scene=image_bgr.copy(), detections=detections, skip_label=True)

segmented_image = mask_annotator.annotate(scene=image_bgr.copy(), detections=detections)
```

output:



Question 5:

```
train_ds = load_voc(split="sbd_train")
eval_ds = load_voc(split="sbd_eval")
```

These lines load the PASCAL VOC dataset's training and evaluation splits using the `keras_cv.datasets.pascal_voc.segmentation` module. Specifically, the `sbd_train` split is loaded into `train_ds` and the `sbd_eval` split is loaded into `eval_ds`.

```
def preprocess_tfds_inputs(inputs):
    def unpackage_tfds_inputs(tfds_inputs):
        return {
            "images": tfds_inputs["image"],
            "segmentation_masks": tfds_inputs["class_segmentation"],
        }

    outputs = inputs.map(unpackage_tfds_inputs)
    outputs = outputs.map(keras_cv.layers.Resizing(height=224, width=224))
    outputs = outputs.batch(32, drop_remainder=True)
    return outputs


train_ds = preprocess_tfds_inputs(train_ds)
batch = train_ds.take(1).get_single_element()
keras_cv.visualization.plot_segmentation_mask_gallery(
    batch["images"],
    value_range=(0, 255),
    num_classes=21,
    y_true=batch["segmentation_masks"],
    scale=3,
    rows=2,
    cols=2,
)
```

1. Defines `preprocess_tfds_inputs` to extract images and segmentation masks, resize them to 224x224, and batch them.
2. Preprocesses the training dataset `train_ds`.
3. Visualizes a batch of images and their segmentation masks using `plot_segmentation_mask_gallery`.

```
eval_ds = preprocess_tfds_inputs(eval_ds)
```

This line preprocesses the evaluation dataset `eval_ds` using the previously defined `preprocess_tfds_inputs` function, which extracts images and segmentation masks, resizes them to 224x224, and batches them with a batch size of 32.

```
train_ds = train_ds.map(keras_cv.layers.RandomFlip())
train_ds = train_ds.map(keras_cv.layers.RandomRotation(factor=.1,segmentation_classes=21))

batch = train_ds.take(1).get_single_element()

keras_cv.visualization.plot_segmentation_mask_gallery(
    batch["images"],
    value_range=(0, 255),
    num_classes=21,
    y_true=batch["segmentation_masks"],
    scale=3,
    rows=2,
    cols=2,
)
```

- Applies random horizontal flipping to the training dataset `train_ds`.
- Applies random rotation to the training dataset `train_ds` with a rotation factor of 0.1 and 21 segmentation classes.
- Takes a batch of data from the augmented `train_ds`.
- Visualizes the augmented images and their segmentation masks using `plot_segmentation_mask_gallery`.

```
def unet_model(input_size=(224,224, 3)):
    #build the model
    ####################
    #your code goes here
    ####################
    inputs = tf.keras.layers.Input(input_size)
    # U-Net architecture implementation
    c1 = Conv2D(64, (3, 3), activation='relu', padding='same')(inputs)
    c1 = Dropout(0.1)(c1)
    c1 = Conv2D(64, (3, 3), activation='relu', padding='same')(c1)
    p1 = MaxPooling2D((2, 2))(c1)

    c2 = Conv2D(128, (3, 3), activation='relu', padding='same')(p1)
    c2 = Dropout(0.1)(c2)
    c2 = Conv2D(128, (3, 3), activation='relu', padding='same')(c2)
    p2 = MaxPooling2D((2, 2))(c2)

    c3 = Conv2D(256, (3, 3), activation='relu', padding='same')(p2)
    c3 = Dropout(0.2)(c3)
    c3 = Conv2D(256, (3, 3), activation='relu', padding='same')(c3)
    p3 = MaxPooling2D((2, 2))(c3)
```

Defines `unet_model` function:
- Input Layer: Shape `(224, 224, 3)`.
- Encoder: Convolutional layers with ReLU, dropout, and max-pooling.
- Decoder: Transposed convolutional layers for upsampling, concatenated with encoder layers, followed by more convolutional layers and dropout.
- Output Layer: Convolutional layer with softmax activation for 21 classes.
- Initializes the model by calling `unet_model()`.

```
dice_loss = sm.losses.DiceLoss()
focal_loss = sm.losses.CategoricalFocalLoss()
total_loss = dice_loss + (1 * focal_loss)
```

This code defines the total loss for the model training by combining Dice loss and Categorical Focal loss equally:

1. Dice Loss for overlap evaluation.
2. Focal Loss to handle class imbalance.
3. Total Loss is the sum of Dice and Focal losses.

This function calculates the Jaccard coefficient (Intersection over Union) for evaluating the performance of segmentation models:

1. Flattening: Flattens the true (`y_true`) and predicted (`y_pred`) segmentation masks.
2. Intersection: Computes the intersection between the true and predicted masks.
3. Jaccard Coefficient: Returns the ratio of the intersection over the union, adding 1.0 for numerical stability.

```
def jaccard_coef(y_true, y_pred):
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    intersection = K.sum(y_true_f * y_pred_f)
    return (intersection + 1.0) / (K.sum(y_true_f) + K.sum(y_pred_f) - intersection + 1.0)
```

I used Adam as optimizer and learning rate .001.

```
metrics=['accuracy', jaccard_coef]
#compile the model
####################
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001), loss=total_loss, metrics=metrics)
####################
```

This code trains the compiled U-Net model:

```
model.fit(train_ds, validation_data=eval_ds, epochs=10 )
```

```
Epoch 1/10
      4/Unknown 670s 165s/step - accuracy: 0.3779 - jaccard_coef: 0.2555 - loss: 1.0459
```

This code:

```
activation='softmax'

LR = 0.001
optim = keras.optimizers.Adam(LR)


dice_loss = sm.losses.DiceLoss()
focal_loss = sm.losses.CategoricalFocalLoss()
total_loss = dice_loss + (1 * focal_loss)

```

1. Sets the activation function to softmax.
2. Defines the learning rate (0.001) and optimizer (Adam).
3. Combines loss functions by adding Dice loss and Categorical Focal loss to create the total loss for training.

```
BACKBONE1 = 'mobilenetv2'

n_classes=21
# define model
model1 = sm.Unet(BACKBONE1, encoder_weights='imagenet', classes=n_classes, activation=activation)
model1.compile(optim, total_loss, metrics=metrics)
print(model1.summary())
```

Sets the backbone for the U-Net model to MobileNetV2 with pre-trained ImageNet weights.
Defines the number of classes (21).
Creates the U-Net model with MobileNetV2 as the encoder.
Compiles the model using the Adam optimizer, total loss (Dice loss + Categorical Focal loss), and specified metrics (accuracy and Jaccard coefficient).

```
flag = True
for l in model1.layers:
  if  l.name=='decoder_stage0_upsampling':
    flag = False
  if flag:
    l.trainable = False
```

Initializes a flag to True.
Iterates through model layers.
Freezes layers (sets trainable = False) until the layer named 'decoder_stage0_upsampling' is reached.
Stops freezing subsequent layers after encountering 'decoder_stage0_upsampling'.

```
model1.fit(train_ds, validation_data=eval_ds, epochs=10, batch_size=32 )
```

```
# Ensure all layers are trainable
for layer in model.layers:
    layer.trainable = True
```

```
LR = 0.000005
optim = keras.optimizers.Adam(LR)
```

```
model1.compile(optim, total_loss, metrics=metrics)
```

```
model1.fit(train_ds, validation_data=eval_ds, epochs=5, batch_size=32 )
```

Initial Training: Trains the model for 10 epochs.
Set All Layers Trainable: Makes all model layers trainable.
Define New Learning Rate and Optimizer: Sets a new learning rate and defines the Adam optimizer.
Recompile the Model: Recompiles the model with the new optimizer and specified loss and metrics.
Continue Training: Trains the model for an additional 5 epochs.