# Technical Note for Future Vision Transport

## Introduction

The objective for this work within the company is to build an initial model for image segmentation that can easily integrate into the complete embedded system chain.

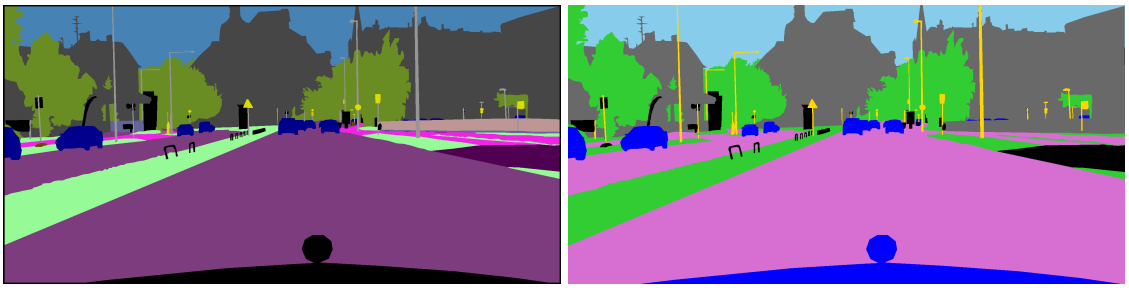
Here, we will look at the different models tested to meet this objective and compare their performances with the aim of constructing a final model that can subsequently be deployed online in the form of an API. We will use Unet Mini, SegNet, as well as Unet and SegNet, both with a pre-trained VGG16 encoder.

For each model, we will attempt to train them on the standard data provided and on augmented data.

## Data presentation

The standard data provided to us are the real images with their masks, as well as a JSON file containing the values of the polygons traced on the masks for the different subcategories. Here, we will retrieve the subcategories and the JSON data to group the 32 subcategories into 8 main categories. This will allow us to simplify the segmentation.

Here is an illustration of the transformation of subcategories into categories :



32 categories | 8 categories

In terms of data training, as we have a lot of data here, we are obliged to load them one by one during training. For this, we will use an ImageDataGenerator. We code in the form of a class so that we can use it for our different data sets (train, validation, and testing).

```
class ImageDataGenerator(Sequence):
    def __init__(self, image_files, mask_files, resize, batch_size, augment=False):
        self.image_files = image_files
        self.mask_files = mask_files
        self.resize = resize
        self.batch_size = batch_size
        self.augment = augment
        if self.augment:
            self.aug = iaa.Sequential([
                iaa.Fliplr(0.4),
                iaa.Affine(scale=(0.8, 1.2)),
                iaa.AdditiveGaussianNoise(scale=0.05*255)
            ])
```

*ImageDataGenerator class code*

```
# Creation of generators
train_generator_aug = ImageDataGenerator(train_image_files, train_mask_files, 256, 4, augment=True)
train_generator = ImageDataGenerator(train_image_files, train_mask_files, 256, 4)
val_generator = ImageDataGenerator(val_image_files, val_mask_files, 256, 4)
test_generator = ImageDataGenerator(test_image_files, test_mask_files, 256, 4)
```

*Code for creating generators*

Furthermore, we integrate the option to choose whether or not to use the data augmentation technique when using the generator. If we use it, then our images will undergo transformations:

- 40% chance that the image will be flipped horizontally
- We have a random scale change factor for the image between 0.8 and 1.2
- Addition of Gaussian noise (5% of the image's value range) This allows us to increase the diversity of our training data and potentially improve the model's ability to generalize.

# Presentation of the tested models and summary of the state of the art

**Precision :**

Here we use the dice coefficient as a metric for our models, which is a measure of similarity between the true mask and the predicted mask, the more the coefficient tends towards 1, the more the similarity is said to be "perfect".

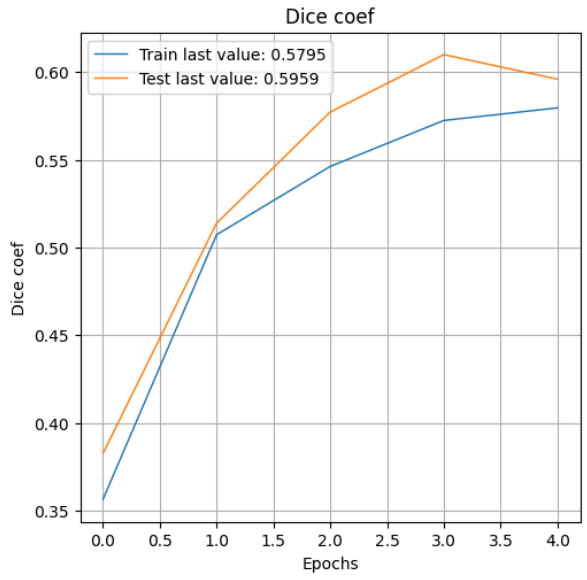
We train all our test models over 4 epochs for reasons of time and computational resources.

## Presentation of the tested models

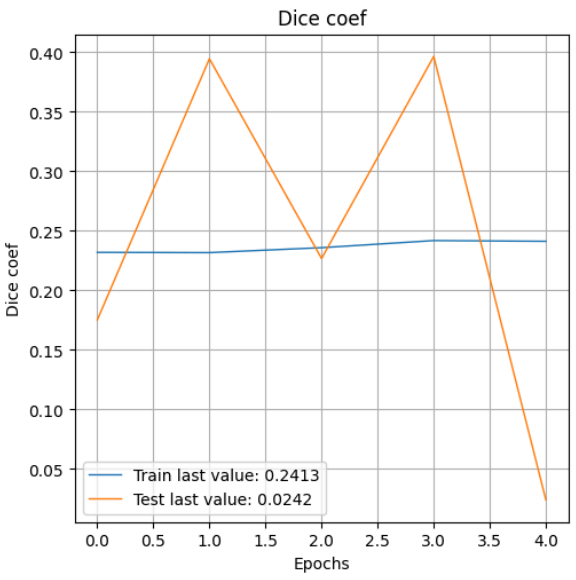
### Unet Mini Model

We will first use the architecture of the Unet Mini model, here we first use an encoder side that will take the image input and shrink it to increase it up to 128 and then there is the decoder part that will concatenate and shrink the shape of the image until it gives it as many dimensions as there are classes (here 8).

```python
def unet_mini(input_shape=(256, 256, 3), num_classes=8):
    inputs = Input(input_shape)

    conv1 = Conv2D(32, (3, 3), activation='relu', padding='same')(inputs)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)

    conv2 = Conv2D(64, (3, 3), activation='relu', padding='same')(pool1)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)

    conv3 = Conv2D(128, (3, 3), activation='relu', padding='same')(pool2)

    up4 = concatenate([UpSampling2D(size=(2, 2))(conv3), conv2], axis=-1)
    conv4 = Conv2D(64, (3, 3), activation='relu', padding='same')(up4)

    up5 = concatenate([UpSampling2D(size=(2, 2))(conv4), conv1], axis=-1)
    conv5 = Conv2D(32, (3, 3), activation='relu', padding='same')(up5)

    output = Conv2D(num_classes, (1, 1), activation='softmax')(conv5)

    model = Model(inputs=[inputs], outputs=[output])
    model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=[dice_coef])

    return model
```




Without Data Augmentation

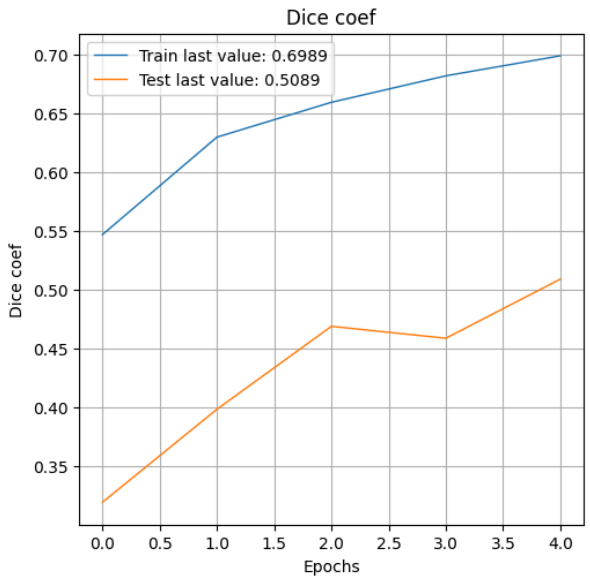



With Data Augmentation

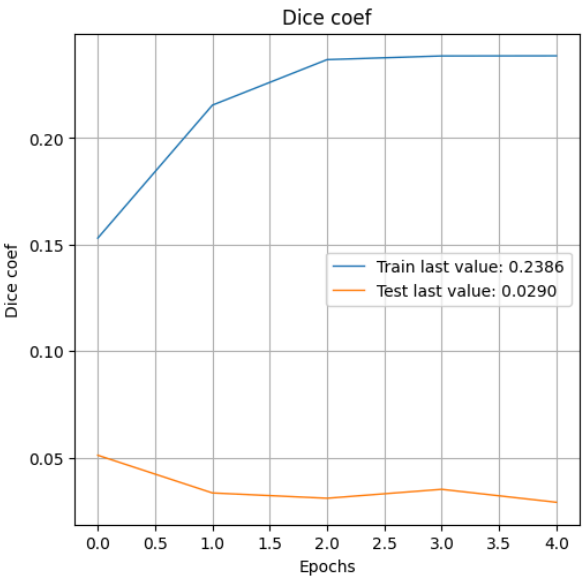
**SegNet Model**

We will try the SegNet model, which differs in its architecture. SegNet focuses on encoding and decoding spatial information using max pooling indices, while Unet emphasizes the preservation of information at different spatial scales through residual connections. This difference in approach can lead to different performance characteristics depending on the specific task and data.

```python
def segnet(input_shape=(256, 256, 3), num_classes=8):
    inputs = Input(input_shape)

    # Encoder
    conv1 = Conv2D(64, (3, 3), activation='relu', padding='same')(inputs)
    conv1 = BatchNormalization()(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)

    conv2 = Conv2D(128, (3, 3), activation='relu', padding='same')(pool1)
    conv2 = BatchNormalization()(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)

    conv3 = Conv2D(256, (3, 3), activation='relu', padding='same')(pool2)
    conv3 = BatchNormalization()(conv3)
    pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)

    # Decoder
    up4 = concatenate([UpSampling2D(size=(2, 2))(conv3), conv2], axis=-1)
    conv4 = Conv2D(128, (3, 3), activation='relu', padding='same')(up4)
    conv4 = BatchNormalization()(conv4)

    up5 = concatenate([UpSampling2D(size=(2, 2))(conv4), conv1], axis=-1)
    conv5 = Conv2D(64, (3, 3), activation='relu', padding='same')(up5)
    conv5 = BatchNormalization()(conv5)

    output = Conv2D(num_classes, (1, 1), activation='softmax')(conv5)

    model = Model(inputs=[inputs], outputs=[output])
    model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=[dice_coef])

    return model
```




Without Data Augmentation

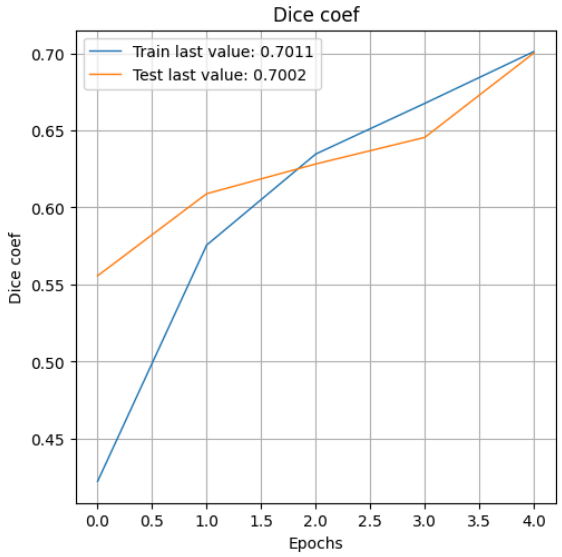



With Data Augmentation

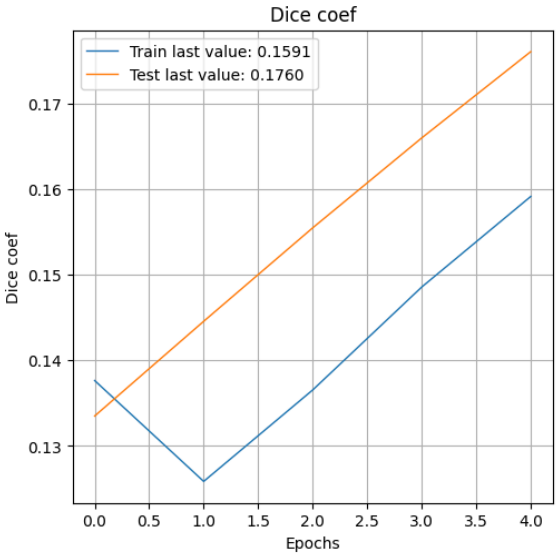
**Unet Model with Pre-trained VGG16 Encoder**

We will test using the pre-trained VGG16 model for the encoding layers, with the ImageNet layers. This will allow us to only have to train our Unet decoder on our data. The use of a pre-trained model can provide a good starting point and can potentially improve the performance by leveraging the knowledge that the model has already learned from a large dataset like ImageNet.

```python
def unet_vgg16(input_shape=(256, 256, 3), num_classes=8):
    # Load pre-trained VGG16 model as encoder
    vgg16 = VGG16(include_top=False, weights='imagenet', input_shape=input_shape)

    # Retrieve outputs from layers of interest
    conv1 = vgg16.get_layer('block1_conv2').output
    conv2 = vgg16.get_layer('block2_conv2').output
    conv3 = vgg16.get_layer('block3_conv3').output
    conv4 = vgg16.get_layer('block4_conv3').output
    conv5 = vgg16.get_layer('block5_conv3').output

    # Create the decoder
    up6 = concatenate([UpSampling2D(size=(2, 2))(conv5), conv4], axis=-1)
    conv6 = Conv2D(512, (3, 3), activation='relu', padding='same')(up6)

    up7 = concatenate([UpSampling2D(size=(2, 2))(conv6), conv3], axis=-1)
    conv7 = Conv2D(256, (3, 3), activation='relu', padding='same')(up7)

    up8 = concatenate([UpSampling2D(size=(2, 2))(conv7), conv2], axis=-1)
    conv8 = Conv2D(128, (3, 3), activation='relu', padding='same')(up8)

    up9 = concatenate([UpSampling2D(size=(2, 2))(conv8), conv1], axis=-1)
    conv9 = Conv2D(64, (3, 3), activation='relu', padding='same')(up9)

    output = Conv2D(num_classes, (1, 1), activation='softmax')(conv9)

    model = Model(inputs=vgg16.input, outputs=output)
    model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=[dice_coef])

    return model
```




Without Data Augmentation

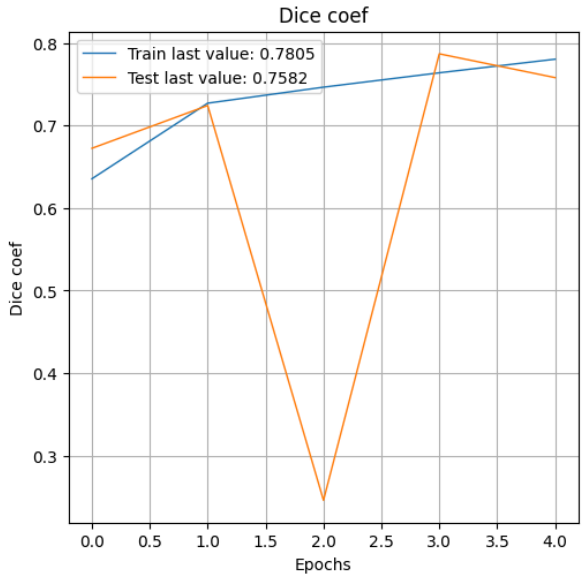



With Data Augmentation

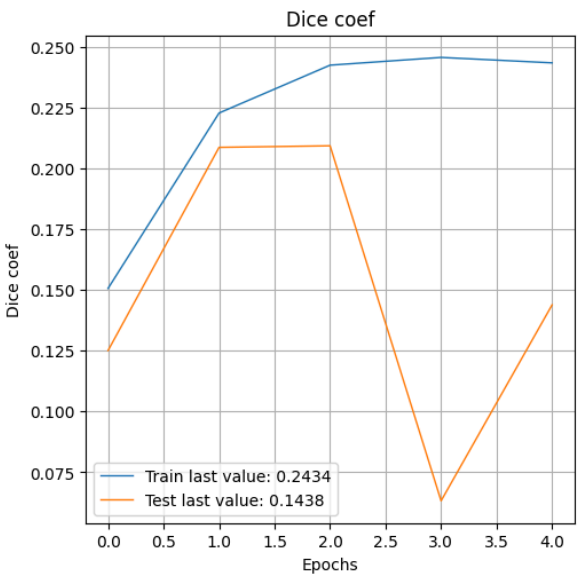
**SegNet Model with Pre-trained VGG16 Encoder**

We will test using SegNet instead of Unet for the decoder layers. This approach allows us to leverage the strengths of SegNet's spatial information encoding and decoding capabilities, while also benefiting from the pre-trained VGG16 encoder. This could potentially improve the model's performance on our specific image segmentation task.

```python
def segnet_vgg16(input_shape=(256, 256, 3), num_classes=8):
    # Load pre-trained VGG16 model as encoder
    vgg16 = VGG16(include_top=False, weights='imagenet', input_shape=input_shape)

    # Retrieve outputs from layers of interest
    conv1 = vgg16.get_layer('block1_conv2').output
    conv2 = vgg16.get_layer('block2_conv2').output
    conv3 = vgg16.get_layer('block3_conv3').output
    conv4 = vgg16.get_layer('block4_conv3').output
    conv5 = vgg16.get_layer('block5_conv3').output

    # Create the decoder
    up6 = concatenate([UpSampling2D(size=(2, 2))(conv5), conv4], axis=-1)
    conv6 = Conv2D(512, (3, 3), activation='relu', padding='same')(up6)
    conv6 = BatchNormalization()(conv6)

    up7 = concatenate([UpSampling2D(size=(2, 2))(conv6), conv3], axis=-1)
    conv7 = Conv2D(256, (3, 3), activation='relu', padding='same')(up7)
    conv7 = BatchNormalization()(conv7)

    up8 = concatenate([UpSampling2D(size=(2, 2))(conv7), conv2], axis=-1)
    conv8 = Conv2D(128, (3, 3), activation='relu', padding='same')(up8)
    conv8 = BatchNormalization()(conv8)

    up9 = concatenate([UpSampling2D(size=(2, 2))(conv8), conv1], axis=-1)
    conv9 = Conv2D(64, (3, 3), activation='relu', padding='same')(up9)
    conv9 = BatchNormalization()(conv9)

    output = Conv2D(num_classes, (1, 1), activation='softmax')(conv9)

    model = Model(inputs=vgg16.input, outputs=output)
    model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=[dice_coef])

    return model
```




Without Data Augmentation




With Data Augmentation

## Models performances review

| Model | Train Time (min) | Dice Coefficient |
|---|---|---|
| Unet Mini | 68.655391 | 0.598732 |
| Unet Mini Aug | 50.144244 | 0.024111 |

| Model | Train Time (min) | Dice Coefficient |
|---|---|---|
| SegNet | 20.004852 | 0.507754 |
| SegNet Aug | 10.624194 | 0.027458 |

| Model | Train Time (min) | Dice Coefficient |
|---|---|---|
| Unet VGG16 | 21.075115 | 0.707642 |
| Unet VGG16 Aug | 10.417263 | 0.176230 |

| Model | Train Time (min) | Dice Coefficient |
|---|---|---|
| SegNet VGG16 | 9.462162 | 0.771238 |
| SegNet VGG16 Aug | 11.186152 | 0.143912 |

It can be observed that data augmentation doesn't work with our models and our data, we lose a lot in performance and it doesn't help with generalization.

## State of the Art Synthesis

Image segmentation is an active research field in computer vision, with many proposed approaches, including convolutional neural network architectures such as U-Net, SegNet, and their variants with a pre-trained VGG16 encoder.

U-Net and U-Net Mini are widely used convolutional neural network architectures for image segmentation.

They are known for their ability to capture information at different spatial scales through residual connections.

SegNet is another popular architecture for image segmentation. It stands out for its use of max pooling indices to encode and decode spatial information.

The use of a pre-trained VGG16 encoder with these architectures is a common approach to improve segmentation performance. VGG16 is a convolutional neural network model that has shown excellent performance on the ImageNet database.

However, despite these advances, image segmentation remains a challenge, particularly in contexts where data are limited or very varied. In this context, techniques such as data augmentation are often used to improve the robustness of models.

However, as you have observed, data augmentation doesn't always guarantee performance improvement and can sometimes even degrade it.

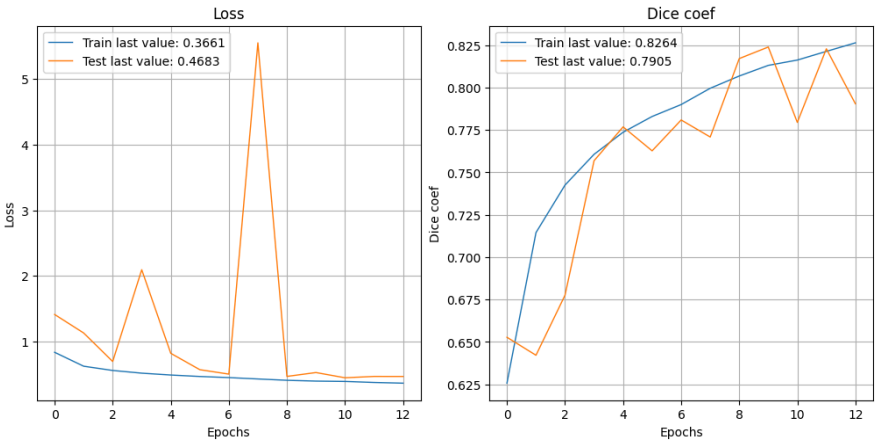
# Detailed presentation of the final model selected

Given the performance of our various models, we are going to choose the one that obtained the best dice coefficient. Therefore, we will select SegNet with the pre-trained VGG16 encoder as our final model.

As this is our final model, we will train it for a longer period with EarlyStopping set to a patience of 3. This means that training will stop if there is no improvement in the dice performance after 3 epochs. We will also use a ModelCheckpoint to save the best weights of our model. This approach ensures that we retain the most effective version of our model, even if the performance fluctuates during training.

```python
def segnet_vgg16(input_shape=(256, 256, 3), num_classes=8):
    # Load pre-trained VGG16 model as encoder
    vgg16 = VGG16(include_top=False, weights='imagenet', input_shape=input_shape)

    # Retrieve outputs from layers of interest
    conv1 = vgg16.get_layer('block1_conv2').output
    conv2 = vgg16.get_layer('block2_conv2').output
    conv3 = vgg16.get_layer('block3_conv3').output
    conv4 = vgg16.get_layer('block4_conv3').output
    conv5 = vgg16.get_layer('block5_conv3').output

    # Create the decoder
    up6 = concatenate([UpSampling2D(size=(2, 2))(conv5), conv4], axis=-1)
    conv6 = Conv2D(512, (3, 3), activation='relu', padding='same')(up6)
    conv6 = BatchNormalization()(conv6)

    up7 = concatenate([UpSampling2D(size=(2, 2))(conv6), conv3], axis=-1)
    conv7 = Conv2D(256, (3, 3), activation='relu', padding='same')(up7)
    conv7 = BatchNormalization()(conv7)

    up8 = concatenate([UpSampling2D(size=(2, 2))(conv7), conv2], axis=-1)
    conv8 = Conv2D(128, (3, 3), activation='relu', padding='same')(up8)
    conv8 = BatchNormalization()(conv8)

    up9 = concatenate([UpSampling2D(size=(2, 2))(conv8), conv1], axis=-1)
    conv9 = Conv2D(64, (3, 3), activation='relu', padding='same')(up9)
    conv9 = BatchNormalization()(conv9)

    output = Conv2D(num_classes, (1, 1), activation='softmax')(conv9)

    model = Model(inputs=vgg16.input, outputs=output)
    model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=[dice_coef])

    return model
```

```python
# Define the callbacks
early_stopping = EarlyStopping(monitor='val_dice_coef', patience=3, mode='max', verbose=1)
model_checkpoint = ModelCheckpoint('best_final_model.h5', monitor='val_dice_coef', mode='max'
```




```
Dice Coefficient of Final Segnet_Vgg16: 0.83
```

# Conclusion and possible avenues for improvement

During this work, we explored several models for image segmentation with the aim of constructing a system that can easily integrate into the complete embedded system chain.

Among the tested models, SegNet with the pre-trained VGG16 encoder obtained the best dice coefficient.
Therefore, we chose this model as our final model.

This work highlighted the importance of model selection and training techniques for image segmentation. Although data augmentation didn't work in this context, it remains a valuable technique that could be useful in other situations.

Moreover, the choice of SegNet with the pre-trained VGG16 encoder as the final model underscores the effectiveness of this architecture for the task of image segmentation.

Finally, this work paves the way for future improvements and explorations in the field of image segmentation, such as exploring other model architectures like Mask R-CNN, PSPNet.

In addition, for improvement, we can train over more epochs and set a higher patience level or even try to train on potentially better data. We can also test using a different metric than the dice, like the IoU (allows the quantification of the degree of overlap between two masks).