

# Tema 6. Segmentación

Ana Jiménez Pastor  
[anjipas@gmail.com](mailto:anjipas@gmail.com)

# Introducción

## Otras tareas de visión por computador

Segmentación  
semántica



CESPED, GATO, ARBOL,  
CIELO

Se "clasifican" los  
píxeles, no los objetos

Segmentación de  
instancias



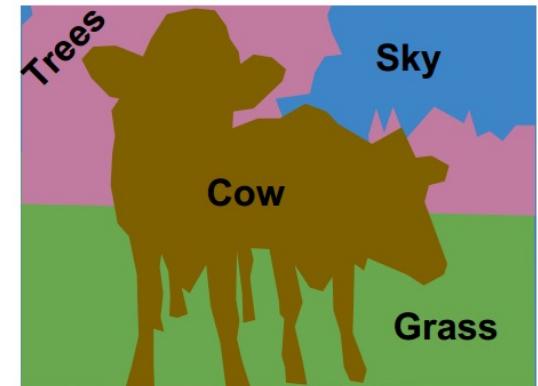
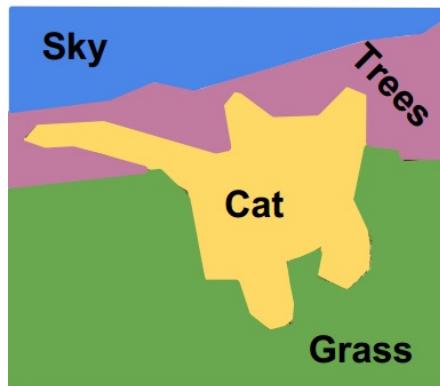
PERRO, PERRO, GATO

Se clasifican los píxeles  
y los objetos.

# Segmentación semántica

# Segmentación semántica

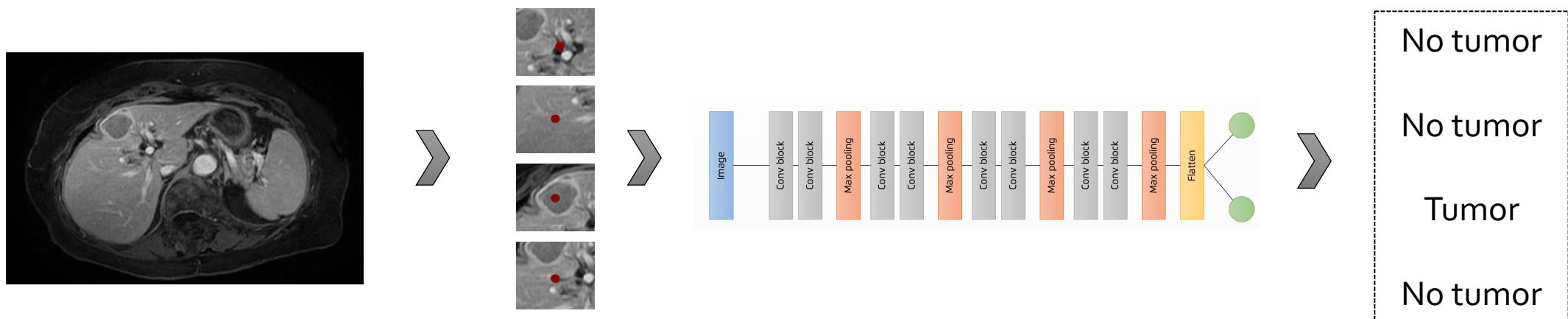
- Se etiqueta cada píxel de la imagen.
- No se diferencian objetos de la misma clase.



# Segmentación semántica

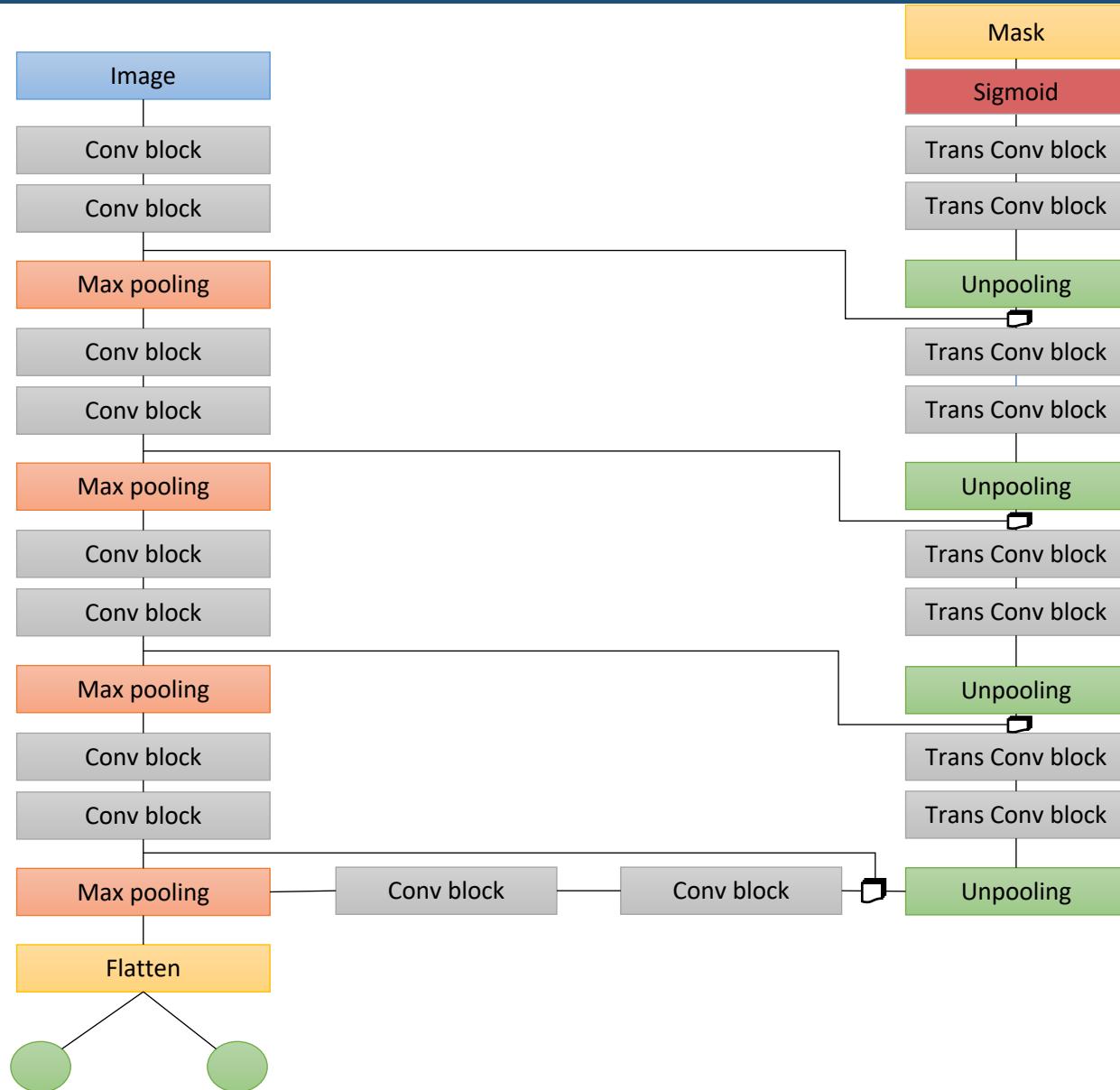
## Solución 1

Clasificación de ventanas de la imagen.



- Cuando la imagen se subdivide en ventanas, **perdemos información del contexto**.
- Clasificación pixel a pixel → Tiempo de **computación elevado**.
- La clasificación de píxeles vecinos es **redundante** dada la similitud entre ellos.

# Segmentación semántica



## U-NET [Ronneberger et al., 2015]

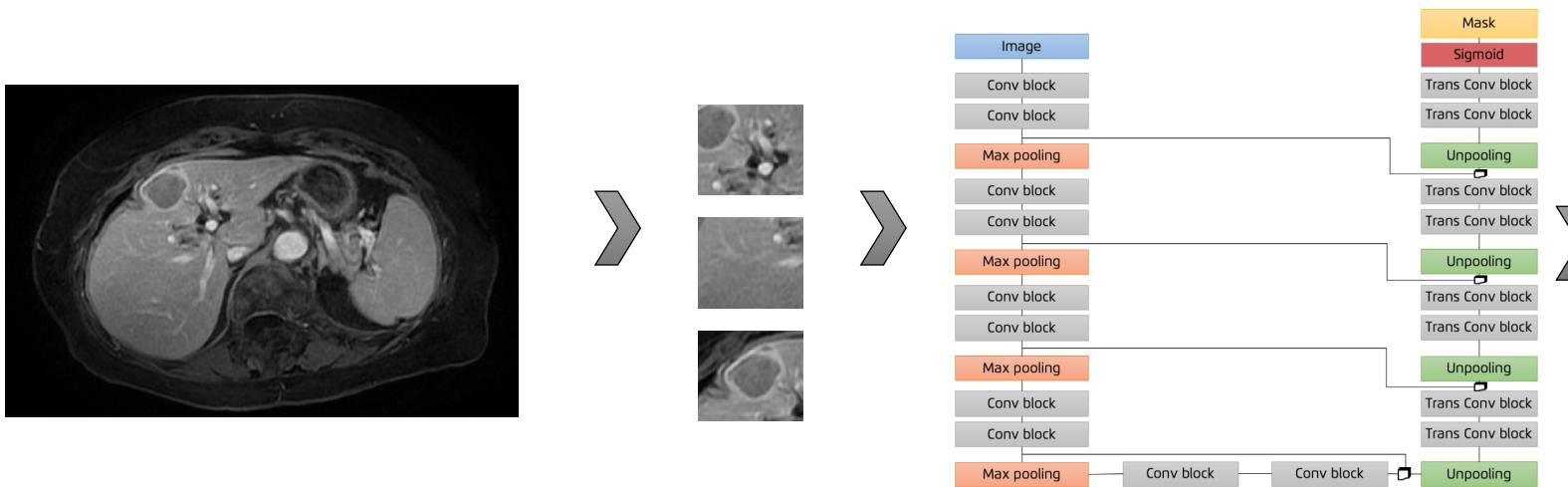
- Red totalmente convolucional (FCN)
- Codificador-decodificador
- Conexiones puente

Ronneberger O, Fischer P, Brox T (2015) U-Net: Convolutional Networks for Biomedical Image Segmentation. In: Navab N., Hornegger J., Wells W., Frangi A. (eds) Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015. MICCAI 2015. Lecture Notes in Computer Science, vol 9351. Springer, Cham

# Segmentación semántica

## Solución 2

Segmentación de ventanas de la imagen.

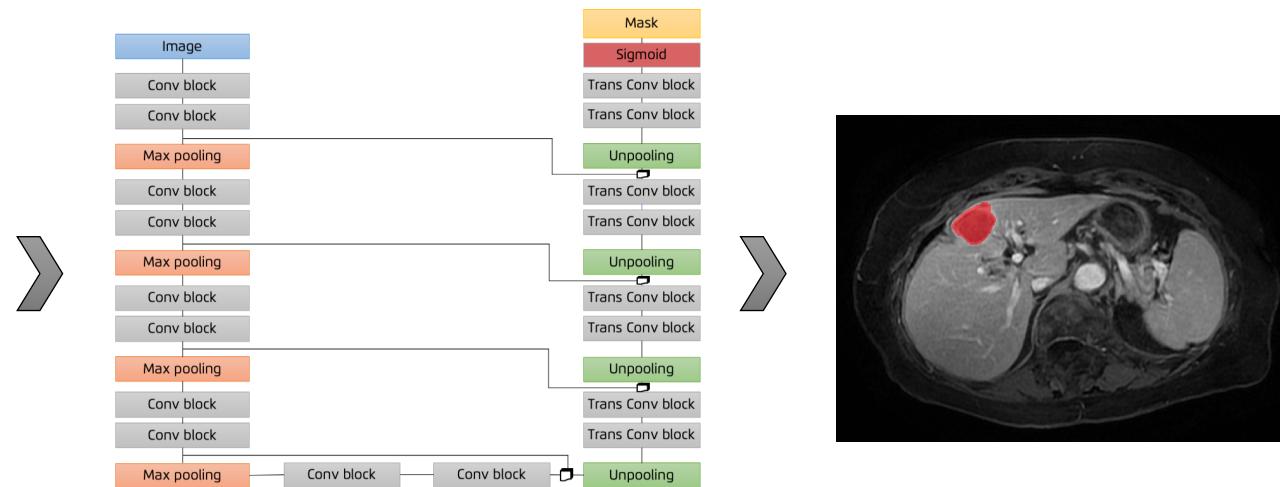
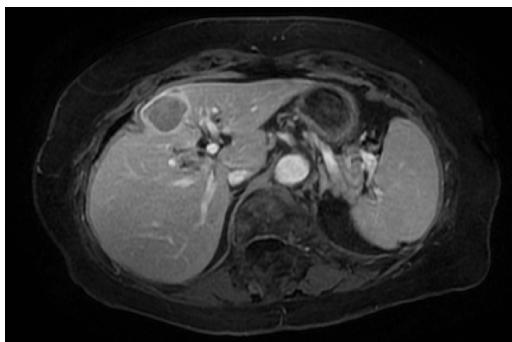


- Reduce el tiempo y coste computacional.
- Elimina la redundancia en la clasificación de píxeles vecinos.
- **Se pierde información contextual.**
- Cada ventana es una muestra independiente.

# Segmentación semántica

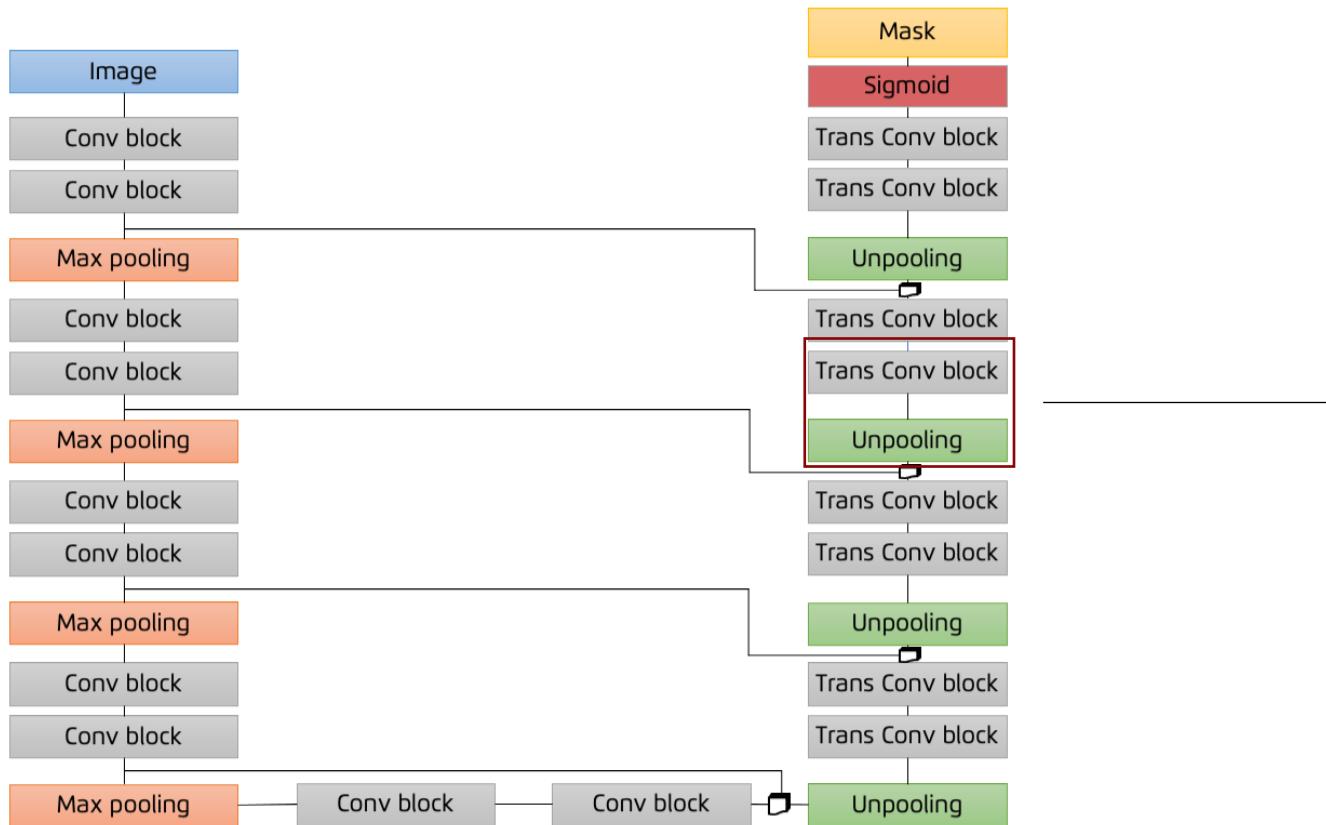
# Solución 3

## Segmentación de imagen completa.



- Reduce el tiempo y coste computacional.
  - Elimina la redundancia en la clasificación de píxeles vecinos.
  - Se preserva la información contextual.
  - Cada imagen es una única muestra de entrenamiento.

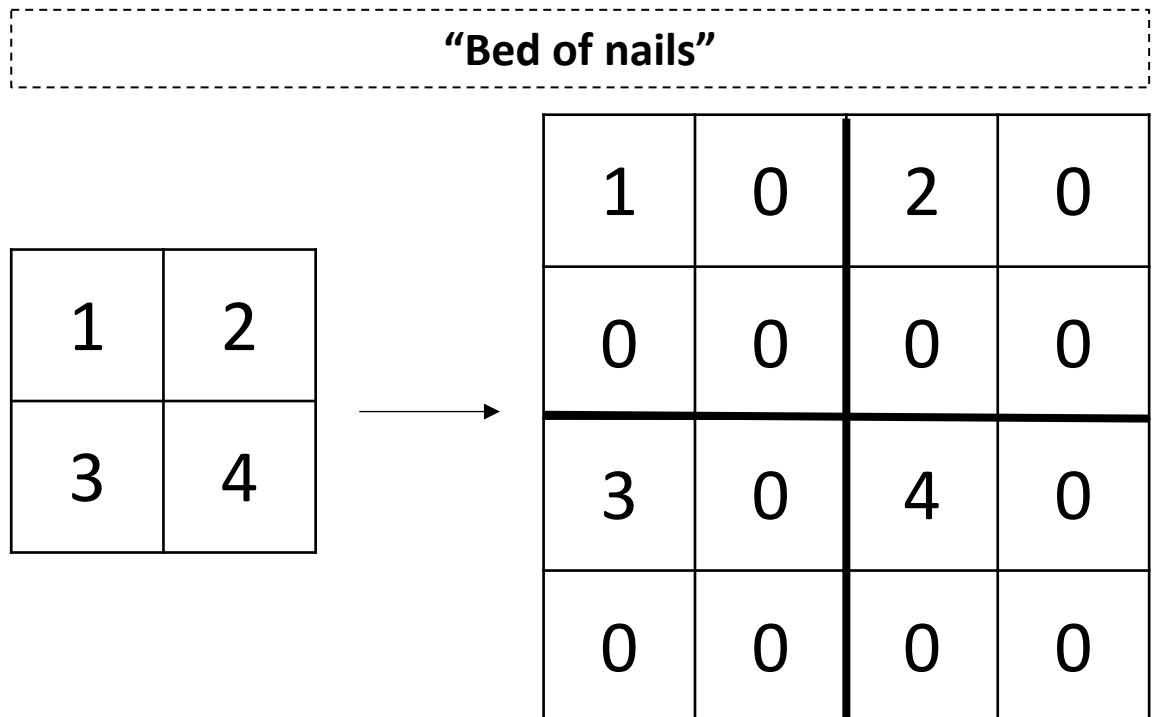
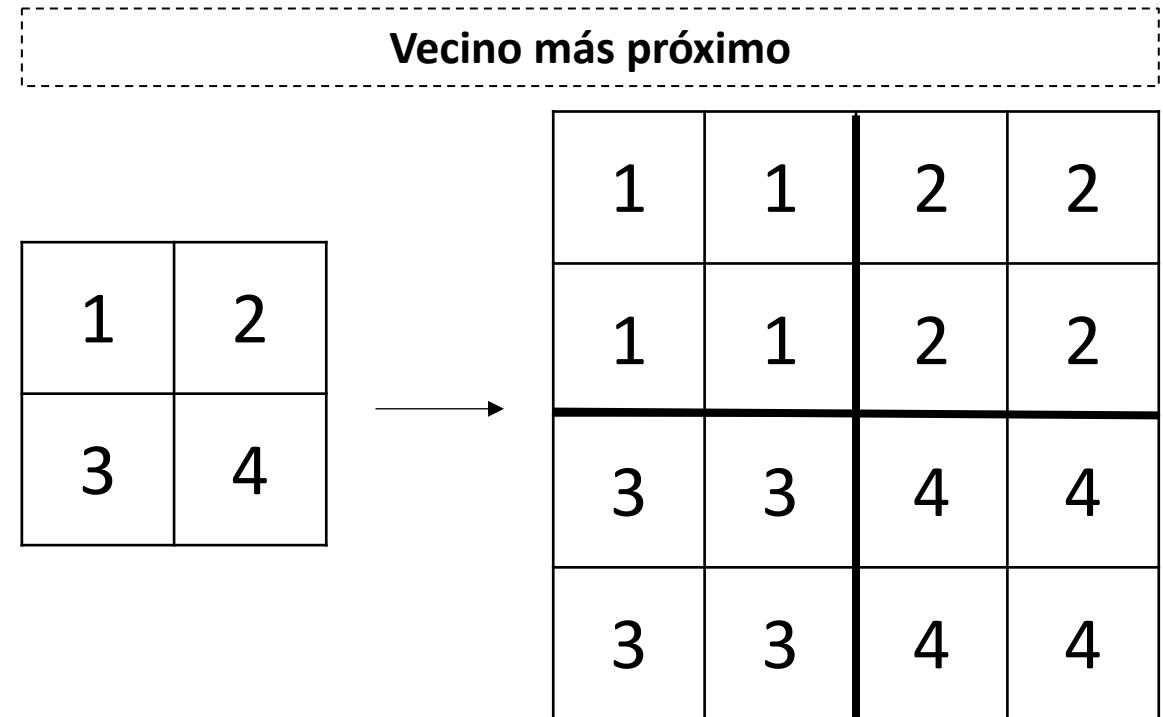
# Segmentación semántica



Capas de *unpooling*  
y convolución  
traspuesta

# Segmentación semántica

## Capa de *unpooling*



# Segmentación semántica

## Capa de *unpooling*

**Max pooling**

Se almacena que elemento fue el máximo

1	2	6	3
3	5	2	1
1	2	2	1
7	3	4	8

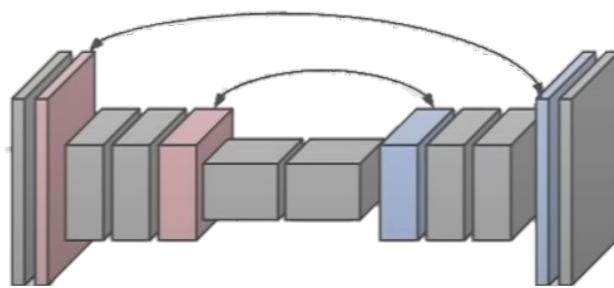
...

5	6
7	8

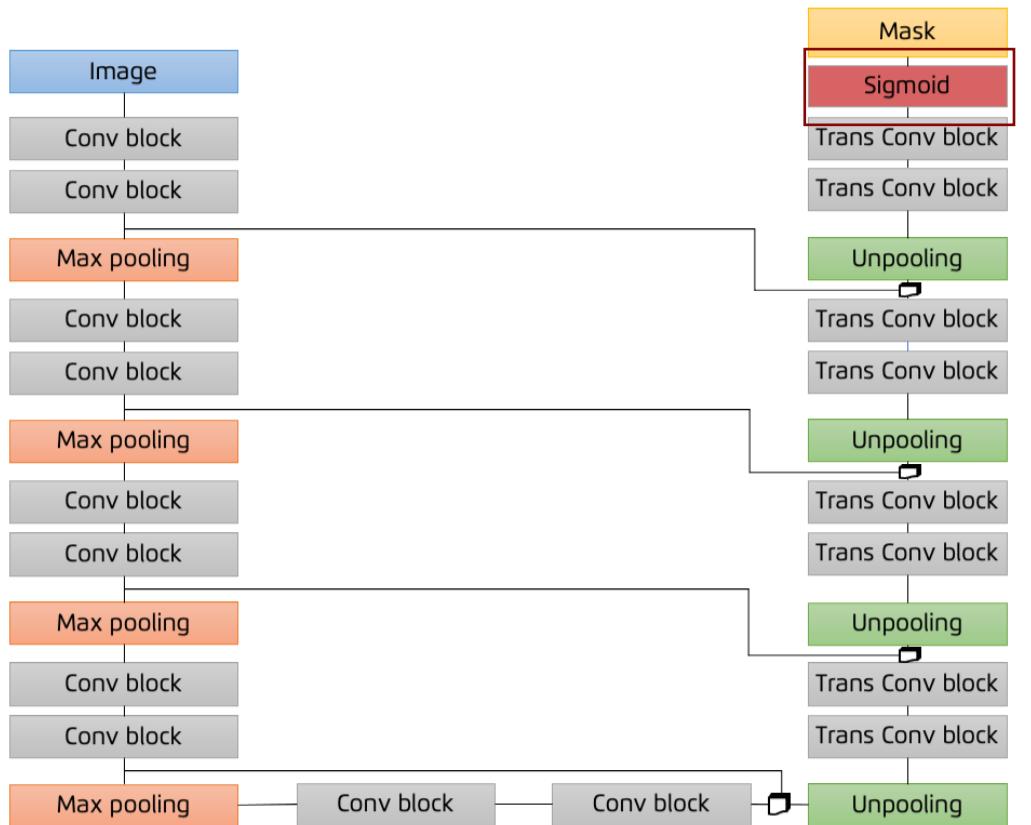
**Max unpooling**

“Bed of nails” pero poniendo el número en la posición del max pooling correspondiente

0	0	2	0
0	1	0	0
0	0	2	0
3	0	0	4



# Segmentación semántica



Función de activación  
de la capa de salida

# Segmentación semántica

## Función de activación de la capa de salida

La función de activación de la capa de salida va a depender del número de clases que queramos segmentar y de las características de estas.

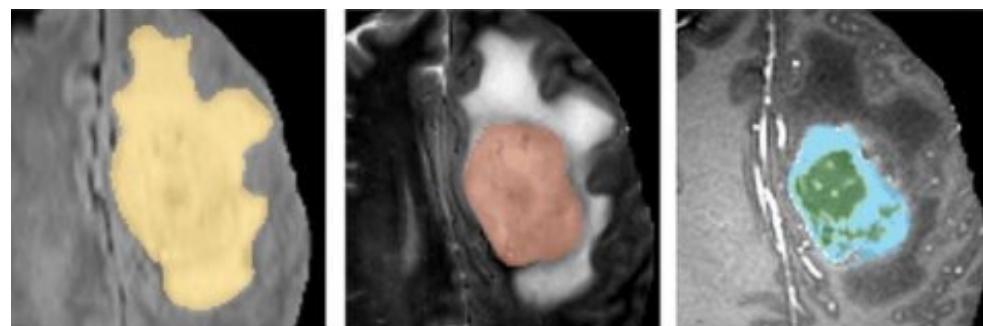
- 2 clases (fondo y región de interés – **segmentación binaria**) → **Sigmoide**
- 2 o más clases (**segmentación multiclase**) donde **no puede** haber solapamiento entre las clases → **Softmax**
- 2 o más clases (**segmentación multiclase**) donde **puede** haber solapamiento entre las clases → **Sigmoide**



Binaria



Multiclase sin solapamiento



Multiclase con solapamiento

# Segmentación semántica

## Función de pérdidas

Se puede emplear la entropía cruzada como en clasificación, pero se ha visto que otras funciones, como las pérdidas Dice, que tienen en cuenta el solapamiento entre las máscaras ofrecen mejores resultados:

- **Pérdidas Dice:** Inversa del coeficiente Dice que mide el grado de solapamiento espacial entre dos máscaras (real y estimada).

$$DC(X, Y) = \frac{2|X \cap Y|}{|X| + |Y|} \quad DCL(X, Y) = 1 - DC(X, Y)$$

# Segmentación semántica

## Métricas

Ecuaciones para evaluar la bondad de la segmentación predicha. Podemos diferenciar 3 grupos principales:

- **Basadas en el solapamiento:** miden la proporción de píxeles correctamente clasificados. Penalizan errores en máscaras pequeñas en mayor medida que en máscaras grandes. Coeficiente Dice, intersección sobre la unión, sensibilidad, especificidad, precisión.
- **Basadas en el área/volumen:** diferencia en el área calculada sobre la máscara de referencia y la calculada sobre la predicha. No consideran la posición de las máscaras. Interesantes cuando el área de la región de interés es importante pero no tanto la segmentación de la forma en general. Diferencia de área absoluta, diferencia de área relativa.
- **Basadas en la distancia:** funciones de la distancia Euclídea entre los píxeles segmentados automáticamente y los píxeles de la máscara de referencia. Distancia media, máxima distancia superficial.

# Segmentación de instancias

# Segmentación de instancias

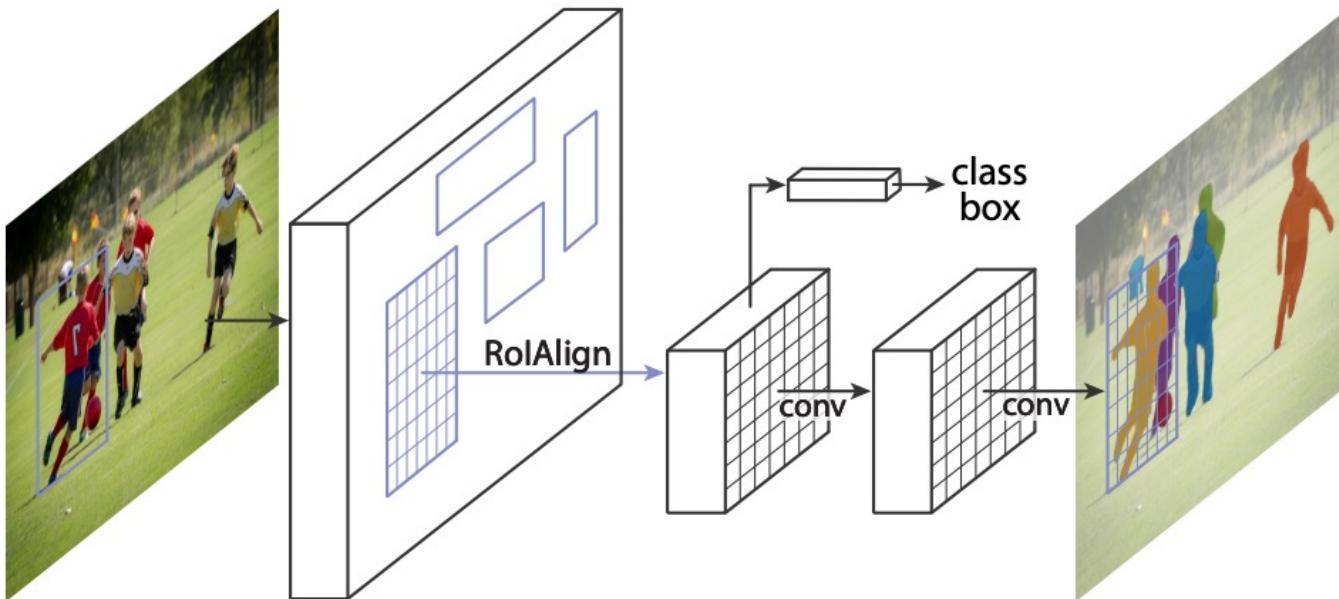
No solo queremos diferenciar los píxeles correspondientes a cada una de las clases, sino además queremos diferenciar instancias de una misma clase.

**Segmentación semántica + detección de objetos**



# Segmentación de instancias

## Mask R-CNN [He et al. 2017]



**Usa como base Faster R-CNN  
(detección de regiones) donde  
además se añade la segmentación  
semántica de la región de interés**

1. Propuesta de regiones con CNNs (como Faster R-CNN).
2. De cada región:
  1. Clasificación y corrección de la ventana.
  2. Clasificación de cada uno de los píxeles de pertenecer o no a la clase clasificada.

# Segmentación de instancias

Mask R-CNN [He et al. 2017]

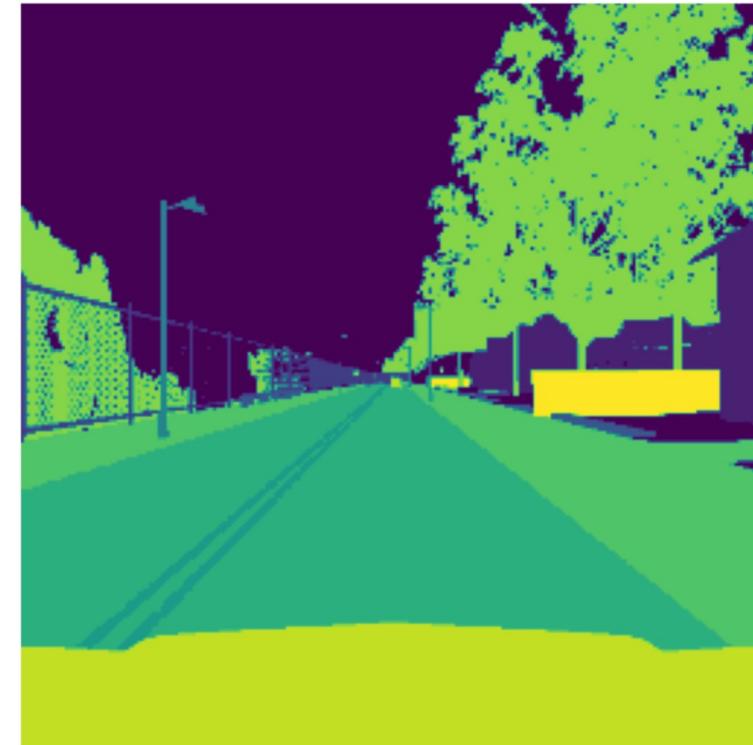


# **Segmentación semántica en TF**

# Segmentación semántica en TF

En un problema de segmentación semántica partimos de 2 matrices:

- **Imagen**: en cada píxel tenemos la intensidad de la imagen.
- **Máscara**: en cada píxel tenemos la clase a la que corresponde cada pixel.



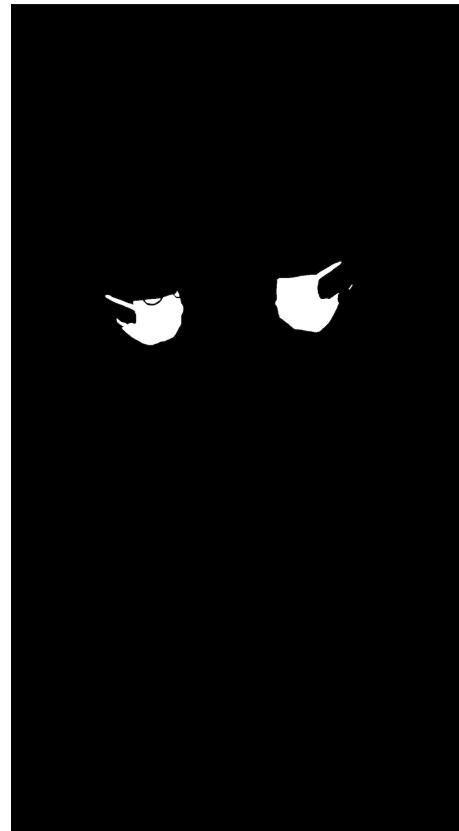
# Segmentación semántica en TF

Dos tipos de segmentación semántica:

- **Binaria:**



Imagen



Máscara

Máscara:

- Matriz de tamaño (**N FILAS, N COLUMNAS, 1**) con valor 1 en la región segmentada y 0 en el fondo.
- Función de activación **sigmoide** en la capa de salida.

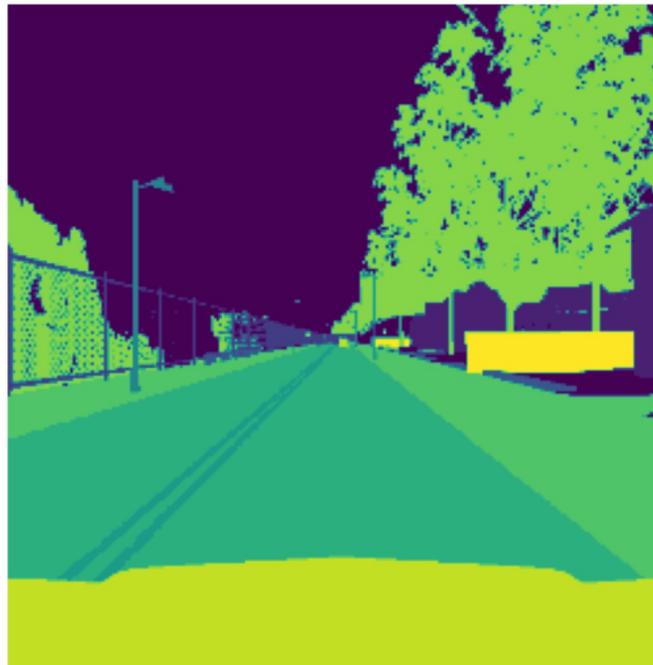
# Segmentación semántica en TF

Dos tipos de segmentación semántica:

- **Multiclasa:**



Imagen



Máscara

Máscara:

- Matriz de tamaño (**N FILAS, N COLUMNAS, N ETIQUETAS**)
- Originalmente tendremos una matriz 2D donde cada pixel tiene el valor de la etiqueta correspondiente (0, 1, 2, ...). A esta se le aplica una **codificación OneHot** para tener, en cada canal, a 1 los píxeles pertenecientes a la clase correspondiente.
- Función de activación **softmax** en la capa de salida.

# Segmentación semántica en TF

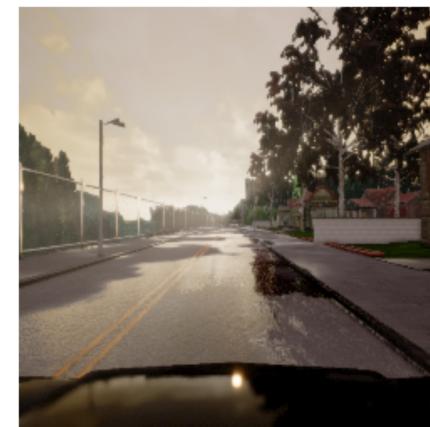
## Segmentación semántica multiclase

Para afrontar este problema vamos a hacer uso de un subconjunto de imágenes de la base de datos del simulador CARLA, empleada para el entrenamiento de modelos de conducción autónoma.

Se trata de un subconjunto de las imágenes que podemos encontrar en el [reto de Kaggle](#).

Vamos a emplear un conjunto de 1000 imágenes (800 entrenamiento, 100 validación, 100 test) en las que tenemos la imagen real y su correspondiente segmentación en 13 clases.

- 0: Fondo
- 1: Edificios
- 2: Vallas
- 3: Otro
- 4: Peatones
- 5: Postes
- 6: Líneas de carretera
- 7: Calles
- 8: Acera
- 9: Vegetación
- 10: Vehículos
- 11: Paredes
- 12: Señales de tráfico



# Segmentación semántica en TF

## Segmentación semántica multiclase

### 1. Creación Datasets

El método "image\_dataset\_from\_directory", que hemos visto anteriormente para crear los *Dataset*, es muy específico para la resolución de problemas de clasificación.

Para un problema de segmentación debemos de crear nosotros mismos los *Dataset*, de manera que contengan tanto las imágenes como las máscaras.

# Segmentación semántica en TF

## Segmentación semántica multiclasé

### 1. Creación Datasets

Creamos un método que, dado el directorio de una imagen y su máscara correspondiente nos devuelva las matrices que necesitamos correspondientes.

```
N_ROWS = 256
N_COLS = 256
N_LABELS = 13

def read_image(image_path, mask_path):
    image = tf.io.read_file(image_path)
    image = tf.image.decode_png(image, channels=3)
    image = tf.image.convert_image_dtype(image, tf.float32)
    image = tf.image.resize(image, (N_ROWS, N_COLS), method='nearest')

    mask = tf.io.read_file(mask_path)
    mask = tf.image.decode_png(mask, channels=3)
    mask = tf.math.reduce_max(mask, axis=-1, keepdims=True)
    mask = tf.image.resize(mask, (N_ROWS, N_COLS), method='nearest')
    mask = tf.squeeze(mask, axis=-1)
    mask = tf.one_hot(tf.cast(mask, tf.int32), N_LABELS)

    return image, mask
```

# Segmentación semántica en TF

## Segmentación semántica multiclase

### 1. Creación Datasets

Ahora, vamos a definir un método que, dada una lista de directorios de imágenes y máscaras correspondientes, nos genere el Dataset.

```
def dataset_generator(image_paths, mask_paths, buffer_size, batch_size):
    image_list = tf.constant(image_paths)
    mask_list = tf.constant(mask_paths)
    dataset = tf.data.Dataset.from_tensor_slices((image_list, mask_list))
    dataset = dataset.map(read_image, num_parallel_calls=tf.data.AUTOTUNE)
    dataset = dataset.cache().shuffle(buffer_size).batch(batch_size)
    return dataset
```

# Segmentación semántica en TF

## Segmentación semántica multiclas

### 1. Creación Datasets

Creamos las listas con los directorios de las imágenes y máscaras de entrenamiento, validación y test.

```
img_dir_train = '/content/dataset_segmentation/dataset_segmentation/CameraRGB/train'  
mask_dir_train = '/content/dataset_segmentation/dataset_segmentation/CameraSeg/train'  
train_image_paths = glob.glob(os.path.join(img_dir_train, '*.png'))  
train_mask_paths = glob.glob(os.path.join(mask_dir_train, '*.png'))  
  
img_dir_val = '/content/dataset_segmentation/dataset_segmentation/CameraRGB/val'  
mask_dir_val = '/content/dataset_segmentation/dataset_segmentation/CameraSeg/val'  
val_image_paths = glob.glob(os.path.join(img_dir_val, '*.png'))  
val_mask_paths = glob.glob(os.path.join(mask_dir_val, '*.png'))  
  
img_dir_test = '/content/dataset_segmentation/dataset_segmentation/CameraRGB/test'  
mask_dir_test = '/content/dataset_segmentation/dataset_segmentation/CameraSeg/test'  
test_image_paths = glob.glob(os.path.join(img_dir_test, '*.png'))  
test_mask_paths = glob.glob(os.path.join(mask_dir_test, '*.png'))
```

# Segmentación semántica en TF

## Segmentación semántica multiclase

### 1. Creación Datasets

Creamos Dataset entrenamiento, validación y test

```
BATCH_SIZE = 32
BUFFER_SIZE = 500

train_dataset = dataset_generator(train_image_paths, train_mask_paths, BUFFER_SIZE, BATCH_SIZE)
validation_dataset = dataset_generator(val_image_paths, val_mask_paths, BUFFER_SIZE, BATCH_SIZE)
test_dataset = dataset_generator(test_image_paths, test_mask_paths, BUFFER_SIZE, BATCH_SIZE)
```

train\_dataset

```
<BatchDataset element_spec=(TensorSpec(shape=(None, 256, 256, 3), dtype=tf.float32, name=None),
TensorSpec(shape=(None, 256, 256, 13), dtype=tf.float32, name=None))>
```

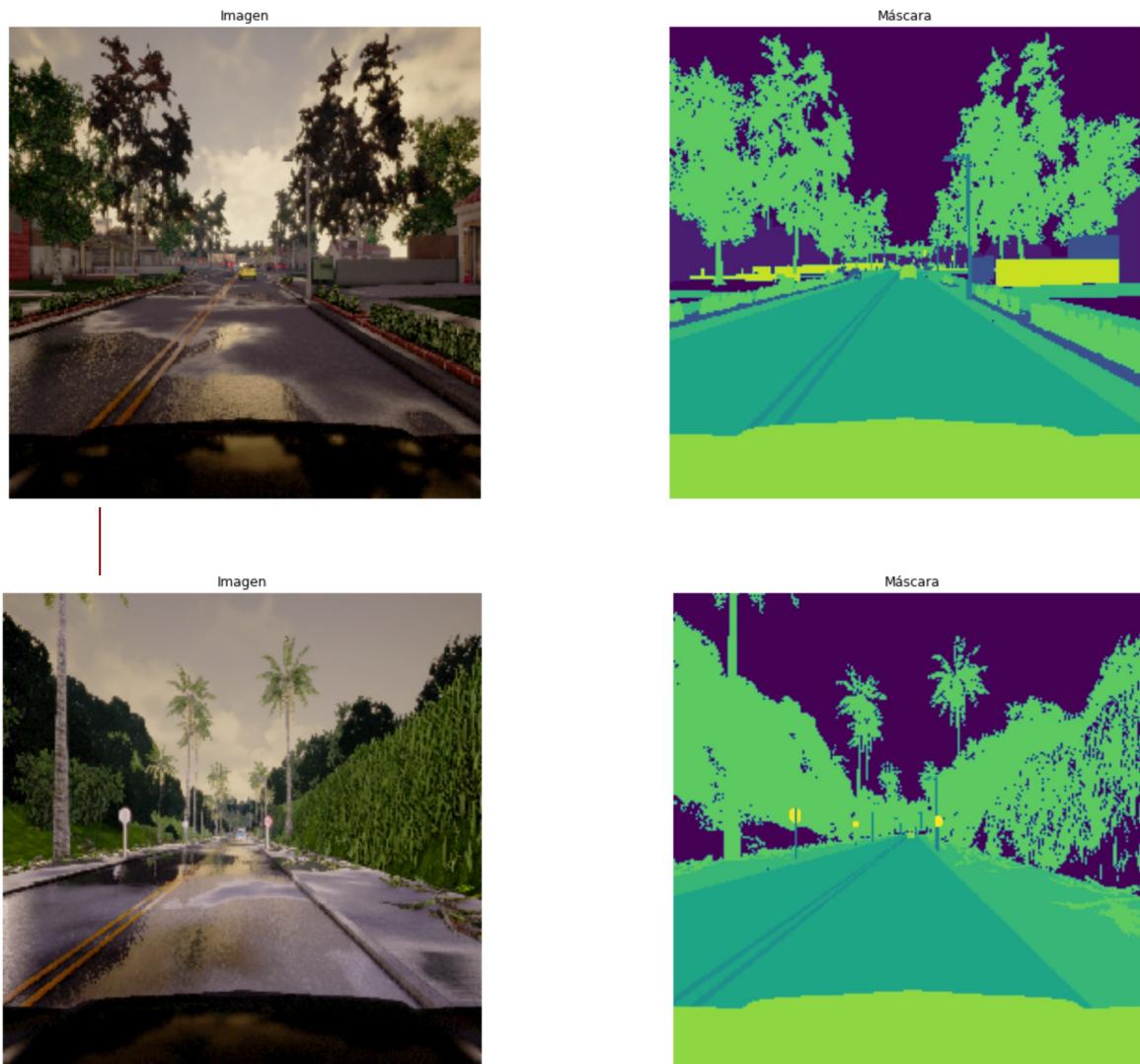
# Segmentación semántica en TF

## Segmentación semántica multiclase

### 1. Creación Datasets

Visualizamos algunos ejemplos (control de calidad)

```
for images, masks in train_dataset.take(1):
    for i in range(3):
        image, mask = images[i], masks[i]
        plt.figure(figsize=(20,8))
        plt.subplot(121)
        plt.imshow(image)
        plt.axis('off')
        plt.title('Imagen')
        plt.subplot(122)
        plt.imshow(tf.math.argmax(mask, axis=-1))
        plt.axis('off')
        plt.title('Máscara')
    plt.show()
```



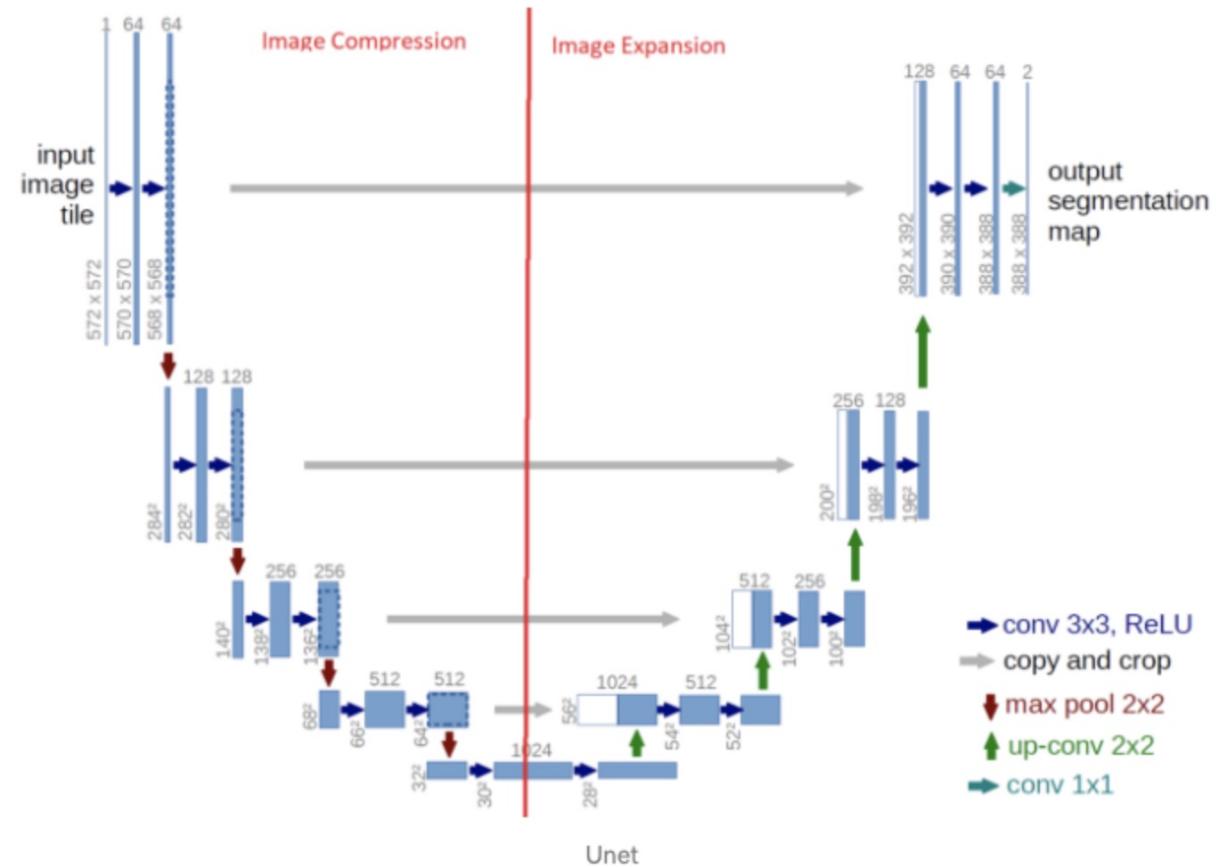
# Segmentación semántica en TF

## Segmentación semántica multiclase

### 2. Definición de la arquitectura

Vamos a emplear la arquitectura UNET.

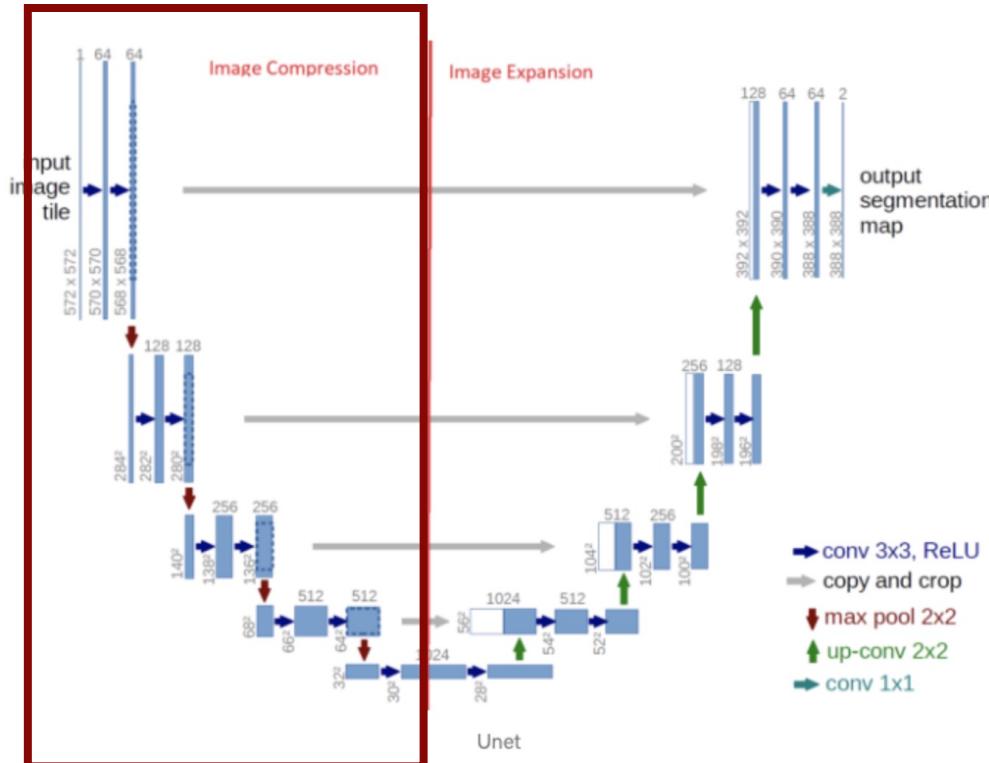
Al tratarse de un problema multiclase, vamos a emplear una función de activación **softmax**.



# Segmentación semántica

## Segmentación semántica multiclasa

### 2. Definición de la arquitectura



```
# Encoding phase
inputs = layers.Input(IMAGE_SHAPE)
conv1 = layers.Conv2D(32, (3, 3), padding='same')(inputs)
conv1 = layers.BatchNormalization()(conv1)
conv1 = layers.Activation('relu')(conv1)
conv1 = layers.Conv2D(32, (3, 3), padding='same')(conv1)
conv1 = layers.BatchNormalization()(conv1)
conv1 = layers.Activation('relu')(conv1)
pool1 = layers.MaxPooling2D(pool_size=(2, 2))(conv1)

conv2 = layers.Conv2D(64, (3, 3), padding='same')(pool1)
conv2 = layers.BatchNormalization()(conv2)
conv2 = layers.Activation('relu')(conv2)
conv2 = layers.Conv2D(64, (3, 3), padding='same')(conv2)
conv2 = layers.BatchNormalization()(conv2)
conv2 = layers.Activation('relu')(conv2)
pool2 = layers.MaxPooling2D(pool_size=(2, 2))(conv2)

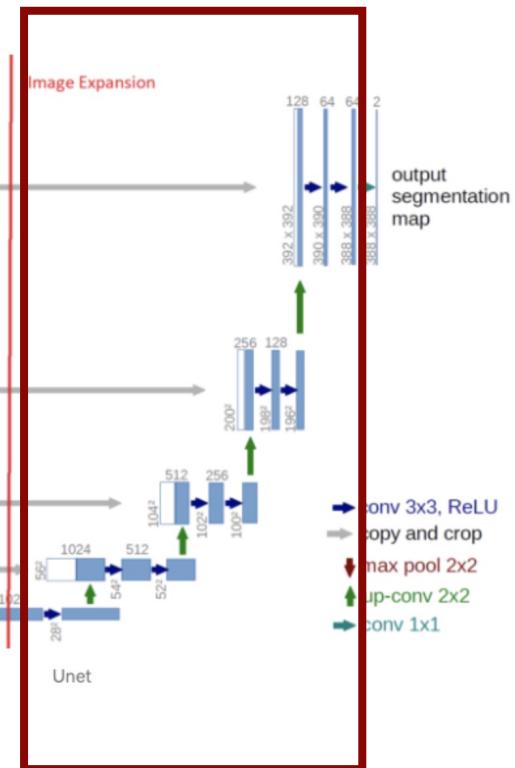
conv3 = layers.Conv2D(128, (3, 3), padding='same')(pool2)
conv3 = layers.BatchNormalization()(conv3)
conv3 = layers.Activation('relu')(conv3)
conv3 = layers.Conv2D(128, (3, 3), padding='same')(conv3)
conv3 = layers.BatchNormalization()(conv3)
conv3 = layers.Activation('relu')(conv3)
pool3 = layers.MaxPooling2D(pool_size=(2, 2))(conv3)

conv4 = layers.Conv2D(256, (3, 3), padding='same')(pool3)
conv4 = layers.BatchNormalization()(conv4)
conv4 = layers.Activation('relu')(conv4)
conv4 = layers.Conv2D(256, (3, 3), padding='same')(conv4)
conv4 = layers.BatchNormalization()(conv4)
conv4 = layers.Activation('relu')(conv4)
pool4 = layers.MaxPooling2D(pool_size=(2, 2))(conv4)
```

# Segmentación

## Segmentación semántica

### 2. Definición de la arquitectura



```
# Decoding phase
conv5 = layers.Conv2D(512, (3, 3), padding='same')(pool4)
conv5 = layers.BatchNormalization()(conv5)
conv5 = layers.Activation('relu')(conv5)
conv5 = layers.Conv2D(512, (3, 3), padding='same')(conv5)
conv5 = layers.BatchNormalization()(conv5)
conv5 = layers.Activation('relu')(conv5)

up6 = layers.concatenate([layers.Conv2DTranspose(256, (2, 2), strides=(2, 2), padding='same')(conv5), conv4], axis=3)
conv6 = layers.Conv2D(256, (3, 3), padding='same')(up6)
conv6 = layers.BatchNormalization()(conv6)
conv6 = layers.Activation('relu')(conv6)
conv6 = layers.Conv2D(256, (3, 3), padding='same')(conv6)
conv6 = layers.BatchNormalization()(conv6)
conv6 = layers.Activation('relu')(conv6)

up7 = layers.concatenate([layers.Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='same')(conv6), conv3], axis=3)
conv7 = layers.Conv2D(128, (3, 3), padding='same')(up7)
conv7 = layers.BatchNormalization()(conv7)
conv7 = layers.Activation('relu')(conv7)
conv7 = layers.Conv2D(128, (3, 3), padding='same')(conv7)
conv7 = layers.BatchNormalization()(conv7)
conv7 = layers.Activation('relu')(conv7)

up8 = layers.concatenate([layers.Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same')(conv7), conv2], axis=3)
conv8 = layers.Conv2D(64, (3, 3), padding='same')(up8)
conv8 = layers.BatchNormalization()(conv8)
conv8 = layers.Activation('relu')(conv8)
conv8 = layers.Conv2D(64, (3, 3), padding='same')(conv8)
conv8 = layers.BatchNormalization()(conv8)
conv8 = layers.Activation('relu')(conv8)

up9 = layers.concatenate([layers.Conv2DTranspose(32, (2, 2), strides=(2, 2), padding='same')(conv8), conv1], axis=3)
conv9 = layers.Conv2D(32, (3, 3), padding='same')(up9)
conv9 = layers.BatchNormalization()(conv9)
conv9 = layers.Activation('relu')(conv9)
conv9 = layers.Conv2D(32, (3, 3), padding='same')(conv9)
conv9 = layers.BatchNormalization()(conv9)
conv9 = layers.Activation('relu')(conv9)
```

# Segmentación semántica en TF

## Segmentación semántica multiclase

### 2. Definición de la arquitectura

Finalmente, se adapta la matriz al número de canales deseados de segmentación (i.e., tantos como clases).

```
out = layers.Conv2D(N_LABELS, (1, 1))(conv9)

# Output
output = layers.Activation(ACTIVATION)(out)

# Compile model with inputs and outputs
model = models.Model(inputs=[inputs], outputs=[output])
```

# Segmentación semántica en TF

## Segmentación semántica multiclase

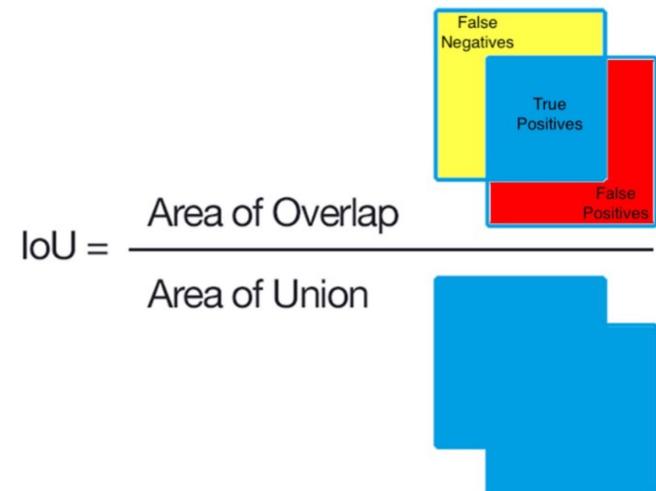
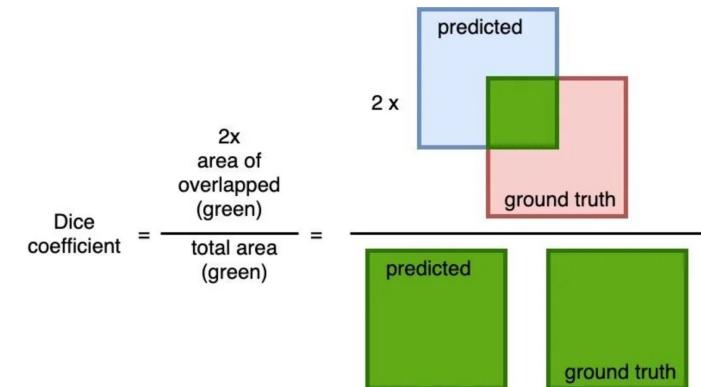
### 3. Entrenamiento y validación

En segmentación, la exactitud no es la métrica más apropiada ya que no tiene en cuenta el solapamiento entre la máscara predicha y la real.

Para evaluar dicho solapamiento es más conveniente emplear métricas como el coeficiente dice (DC) o la intersección sobre la unión (IoU)

$$DC = \frac{2|X \cap Y|}{|X| + |Y|} = \frac{2TP}{2TP + FP + FN}$$

$$IoU = \frac{|X \cap Y|}{|X \cup Y|} = \frac{TP}{TP + FP + FN}$$



# Segmentación semántica en TF

## Segmentación semántica multiclase

### 3. Entrenamiento y validación

$$\text{Dice coefficient} = \frac{2x \text{ area of overlapped (green)}}{\text{total area (green)}} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

```
# smooth para evitar dividir por 0
smooth = 1.
```

```
def dice_coef(targets, inputs):
    inputs = K.flatten(inputs)
    targets = K.flatten(targets)
    intersection = K.sum(targets * inputs)
    dice = (2*intersection + smooth) /
        (K.sum(targets) + K.sum(inputs) + smooth)
    return dice
```

```
def iou(targets, inputs):
    inputs = K.flatten(inputs)
    targets = K.flatten(targets)
    intersection = K.sum(targets * inputs)
    total = K.sum(targets) + K.sum(inputs)
    union = total - intersection
    IoU = (intersection + smooth) / (union +
        smooth)
    return IoU
```

# Segmentación semántica en TF

## Segmentación semántica multiclase

### 3. Entrenamiento y validación

En un problema multiclase, lo que haremos es calcular el coeficiente DICE y la intersección sobre la unión media a lo largo de todas las etiquetas.

```
def dice_coef_multiclass(targets, inputs):  
  
    sum_dice = 0  
    for i in range(N_LABELS):  
        sum_dice += dice_coef(targets[:, :, :, :, i],  
                             inputs[:, :, :, :, i])  
  
    mean_dice = sum_dice / N_LABELS  
    return mean_dice
```

```
def iou_multiclass(targets, inputs):  
  
    sum_iou = 0  
    for i in range(N_LABELS):  
        sum_iou += iou(targets[:, :, :, :, i],  
                      inputs[:, :, :, :, i])  
  
    mean_iou = sum_iou / N_LABELS  
    return mean_iou
```

# Segmentación semántica en TF

## Segmentación semántica multiclase

### 3. Entrenamiento y validación

```
model_unet.compile(optimizer='adam',
                    loss='categorical_crossentropy',
                    metrics=['accuracy', dice_coef_multiclass, iou_multiclass])
```

```
EPOCHS = 50
```

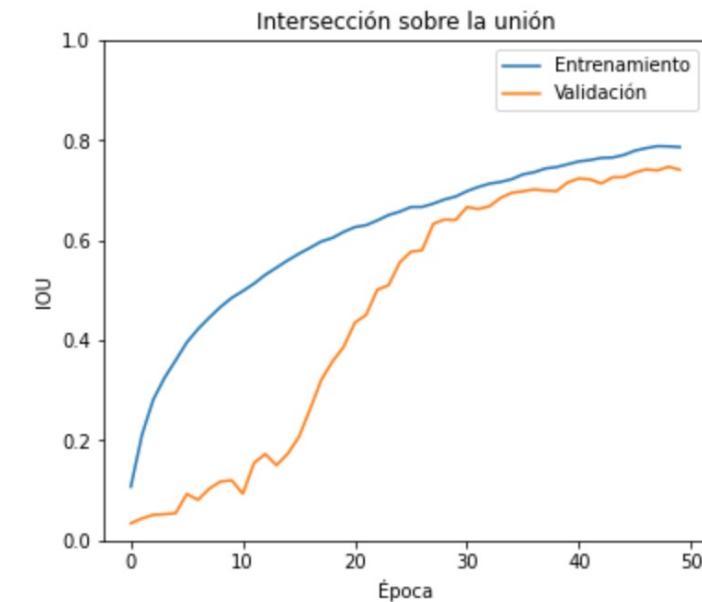
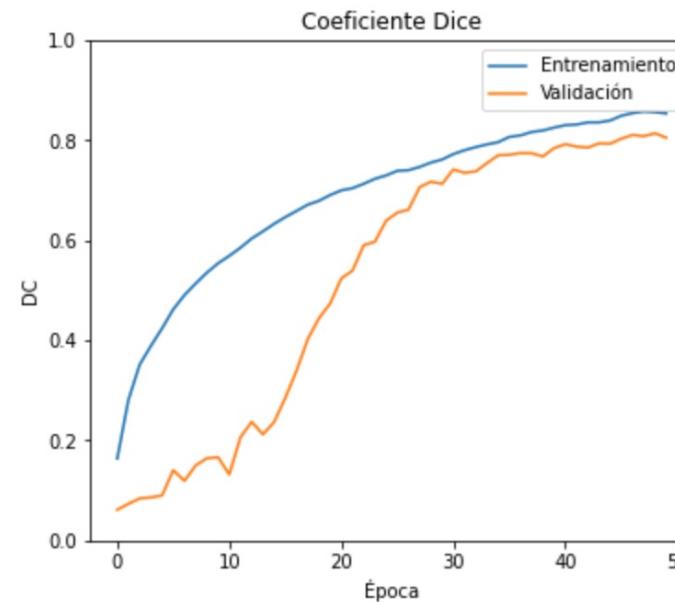
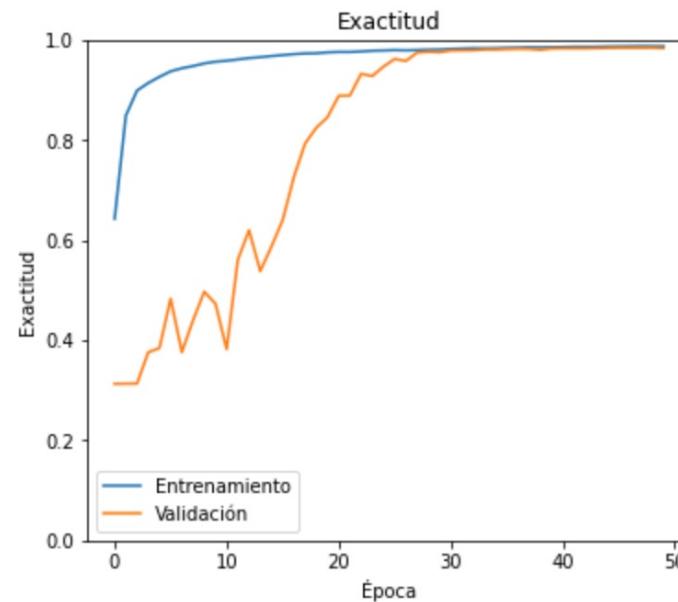
```
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=path_experiment / 'model.h5',
    monitor='val_dice_coef_multiclass',
    mode='max',
    save_best_only=True,
    verbose=1)
```

```
history_unet = model_unet.fit(train_dataset,
                               epochs=EPOCHS,
                               validation_data=validation_dataset,
                               callbacks=[model_checkpoint_callback],
                               verbose=1)
```

# Segmentación semántica en TF

## Segmentación semántica multiclase

### 3. Entrenamiento y validación

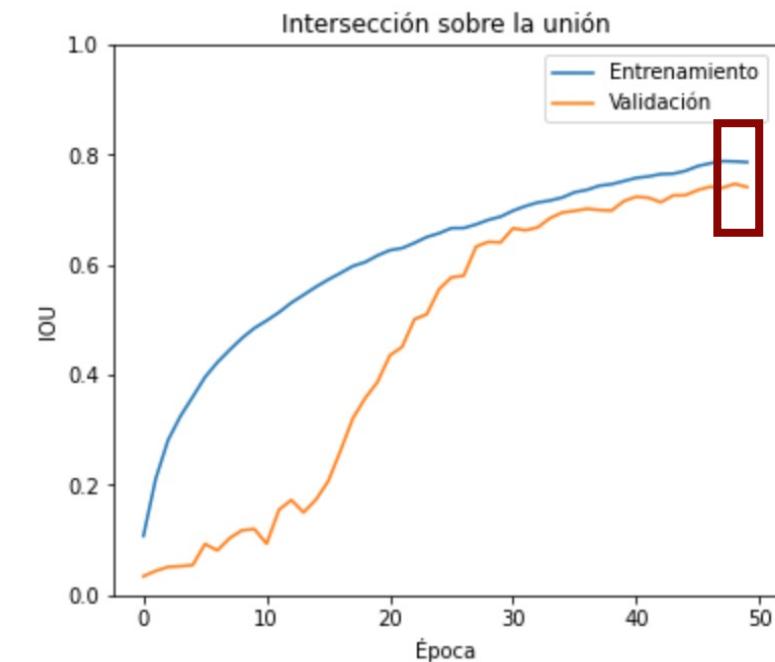
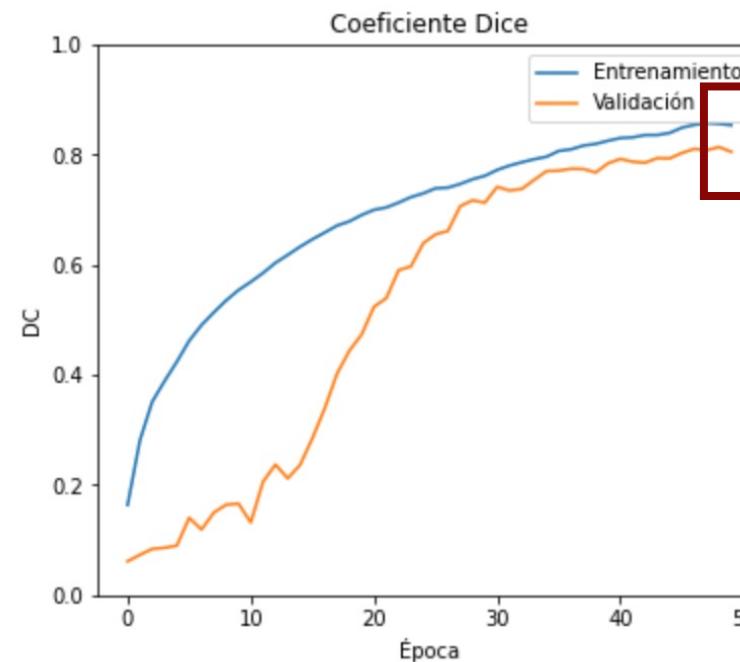
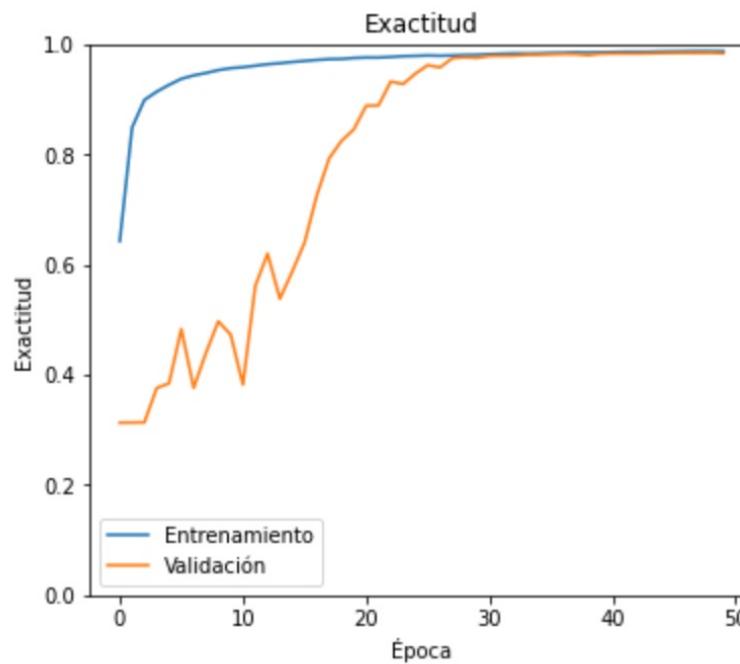


Si sólo hubiéramos empleado la exactitud como métrica, hubiéramos pensado que el modelo ya ha convergido en la época 30 aproximadamente. Pero si observamos el valor de coeficiente Dice o Intersección sobre la unión en esta época, vemos que el método aún no ha convergido.

# Segmentación semántica en TF

## Segmentación semántica multiclase

### 3. Entrenamiento y validación

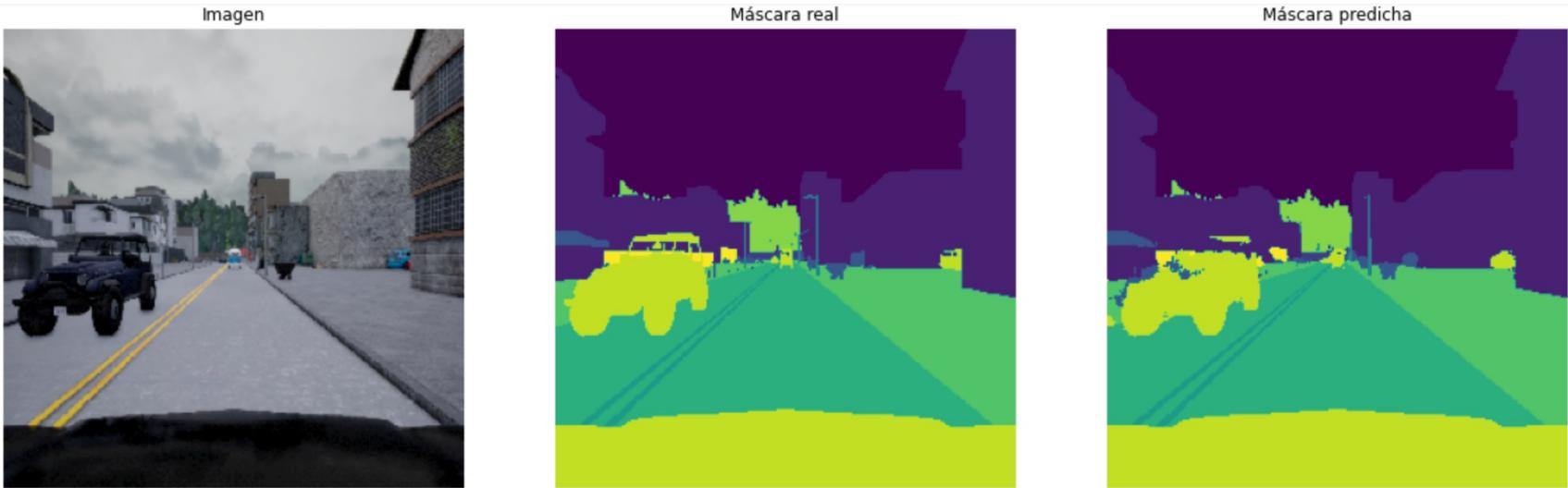


Todavía podríamos haber entrenado durante más épocas.

# Segmentación semántica en TF

## Segmentación semántica multiclase

### 4. Test



# Segmentación semántica en TF

## Segmentación semántica multiclas

### 4. Test

```
# Realizamos la predicción sobre el set de test
model_unet = tf.keras.models.load_model(path_experiment / 'model.h5',
                                         custom_objects={"dice_coef_multiclass": dice_coef_multiclass,
                                                          "iou_multiclass":iou_multiclass})
pred_masks = model_unet.predict(test_dataset)
pred_labels = np.argmax(pred_masks, axis=-1)

# Extraemos las matrices de las máscaras de referencia del Dataset de test para
# calcular las métricas
test_labels = np.empty(shape=[0, 256, 256])
for _, masks in test_dataset.take(-1): # cogemos todos los elementos del dataset
    labels = masks.numpy()
    labels = np.argmax(labels, axis=-1)
    test_labels = np.concatenate((test_labels, labels), axis=0)

print(test_labels.shape)
```

# Segmentación semántica en TF

## Segmentación semántica multiclase

### 4. Test

Para cada una de las clases vamos a calcular las siguientes métricas:

- Coeficiente Dice (o F1-Score)
- Intersección sobre la unión (también conocida como coeficiente Jaccard)
- Precisión

$$Precision = \frac{VP}{VP + FP}$$

- Sensibilidad (o Recall)

$$Sensibilidad = \frac{VP}{VP + FN}$$

- Especificidad

$$Especificidad = \frac{VN}{VN + FP}$$

# Segmentación semántica en TF

## Segmentación semántica multiclase

### 4. Test

index	dc	iou	precision	recall	specificity	label
0	0.8179912248470832	0.6920348156816852	0.8197452781147115	0.8162446620368373	0.9017543031190235	Fondo
1	0.3790039747486556	0.23380931775566133	0.4548260381593715	0.3248496993987976	0.9679086975192416	Edificios
2	0.05558840922531047	0.028588807785888078	0.05465116279069768	0.056558363417569195	0.9874352832084073	Vallas
3	0.02327173169062286	0.011772853185595568	0.1717171717171717	0.012481644640234948	0.9987222239536261	Otro
4	0.0	0.0	0.0	0.0	1.0	Peatones
5	0.0	0.0	0.0	0.0	0.999157862501914	Postes
6	0.0	0.0	0.0	0.0	0.991607666015625	Lineas
7	0.8081124466543989	0.6780106431903291	0.775998195353034	0.8429995098840058	0.9441025256126393	Calles
8	0.5014506769825918	0.3346240722813811	0.585544889892716	0.43847780126849895	0.9623209732893003	Aceras
9	0.2668624233526211	0.15397647996810843	0.21470261256253473	0.35249828884325807	0.9075924320965447	Vegetación
10	0.9970633617042457	0.9941439205955335	0.9941439205955335	1.0	0.9989373198847262	Vehículos
11	0.027935069837674593	0.014165390505359877	0.025676613462873005	0.030629139072847682	0.9781743564233304	Paredes
12	0.0	0.0	0.0	0.0	1.0	Señales
13	0.6817458243246454	0.5171580996323264	0.5854006854904521	0.8160514191456686	0.7881138567406945	Fondo
14	0.0	0.0	0.0	0.0	0.9225616455078125	Edificios
15	0.0	0.0	0.0	0.0	0.9975083691282349	Vallas
16	0.0	0.0	0.0	0.0	0.9886627197265625	Otro
17	0.0	0.0	0.0	0.0	1.0	Peatones
18	0.0	0.0	0.0	0.0	0.9915227054025569	Postes
19	0.0539568345323741	0.027726432532347505	0.05928853754940711	0.04950495049504951	0.9926690281841983	Lineas
20	0.9017249450826461	0.8210374450783658	0.8689013810812687	0.937125748502994	0.963795614842073	Calles
21	0.6690923792197291	0.50273390036452	0.6438436101925695	0.6964022722491058	0.96987644571673	Aceras
22	0.28581261292265336	0.16673358763300541	0.7770760233918129	0.1751093774708766	0.9795339847524965	Vegetación
23	0.9968152866242038	0.9936507936507937	0.9955272835702217	0.9981066268061783	0.9991892037981298	Vehículos
24	0.0	0.0	0.0	0.0	0.99847412109375	Paredes

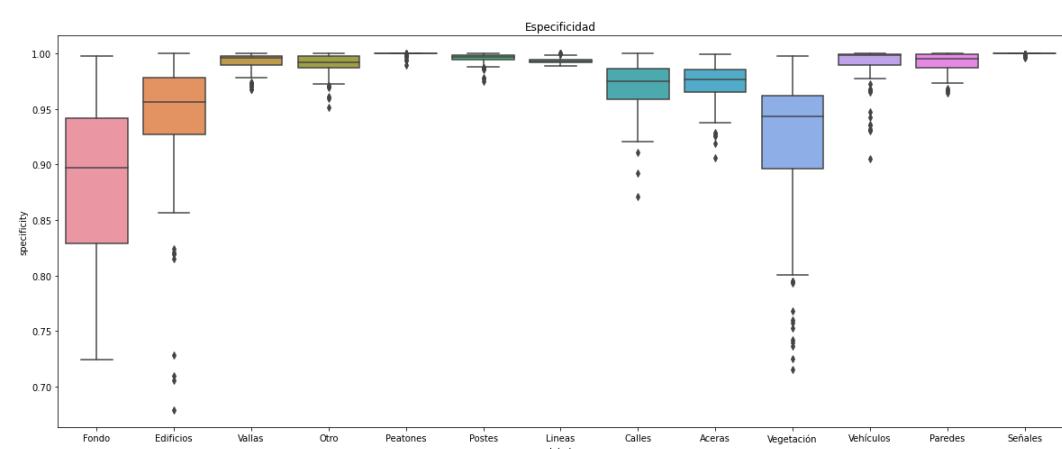
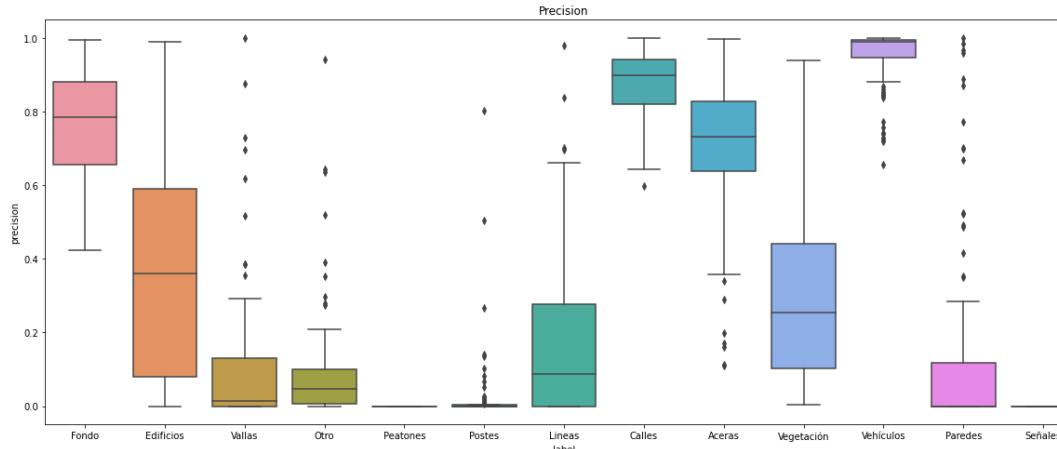
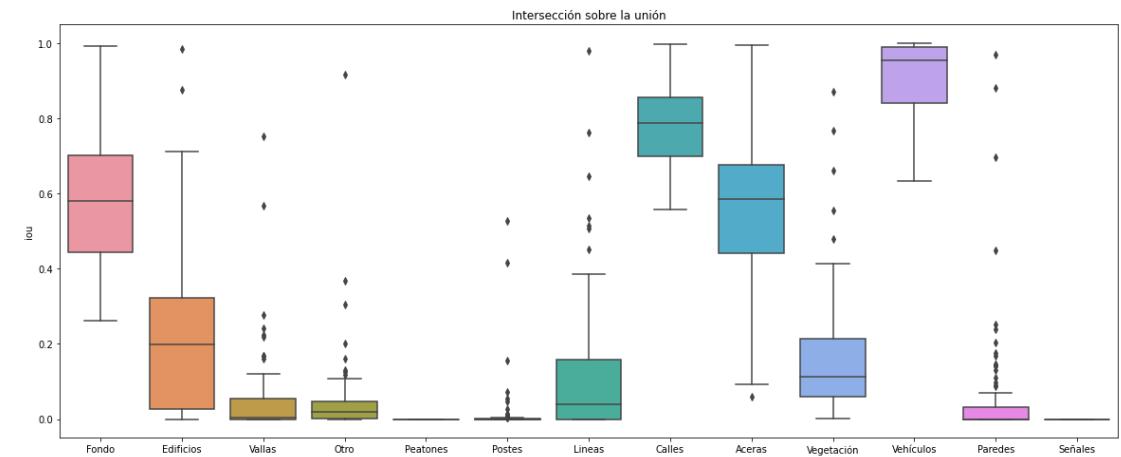
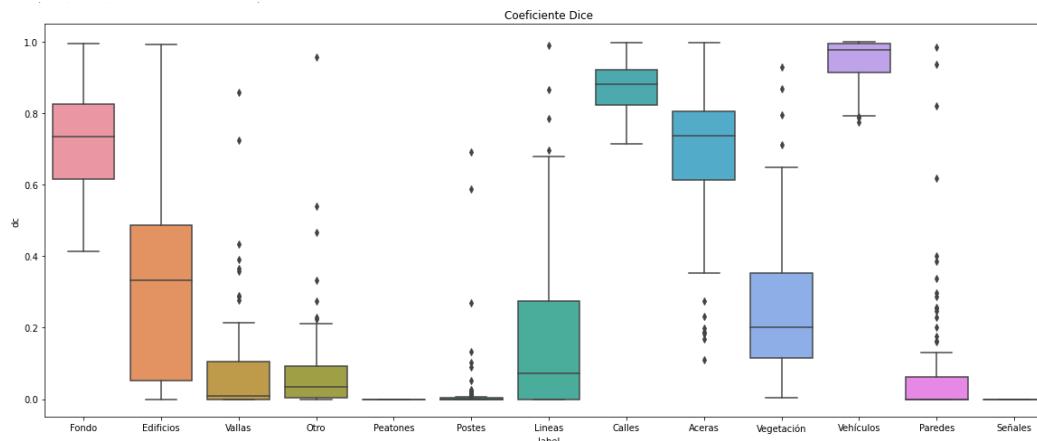
Show 25 ▾ per page

1 2 10 50 52

# Segmentación semántica en TF

## Segmentación semántica multiclas

### 4. Test



# Segmentación semántica en TF

## Segmentación semántica multiclase

### 4. Test

Métricas clase Fondo  
Coeficiente DICE: 0.7445481226378977  
IOU: 0.6076527262725329  
Precisión: 0.7627716760160692  
Sensibilidad: 0.7599752607615762  
Especificidad: 0.8885983049241196

Métricas clase Edificios  
Coeficiente DICE: 0.3279693780147119  
IOU: 0.22261665810914333  
Precisión: 0.370771687743495  
Sensibilidad: 0.3681171901073364  
Especificidad: 0.9384466727790515

Métricas clase Vallas  
Coeficiente DICE: 0.07989052419714532  
IOU: 0.047797723274734055  
Precisión: 0.11768148041283091  
Sensibilidad: 0.12790823746694033  
Especificidad: 0.9930429896136955

Métricas clase Otro  
Coeficiente DICE: 0.06609596826556548  
IOU: 0.03699537140405901  
Precisión: 0.08559762194219347  
Sensibilidad: 0.09229690504452316  
Especificidad: 0.9900765618077159

Métricas clase Peatones  
Coeficiente DICE: 0.0  
IOU: 0.0  
Precisión: 0.0  
Sensibilidad: 0.0  
Especificidad: 0.9996816432310027

Métricas clase Postes  
Coeficiente DICE: 0.020722173205538076  
IOU: 0.012314248131124902  
Precisión: 0.02121728619340582  
Sensibilidad: 0.023995920672687063  
Especificidad: 0.9950191130522845

Métricas clase Lineas  
Coeficiente DICE: 0.15654703462149316  
IOU: 0.1003605066902042  
Precisión: 0.158685243107055  
Sensibilidad: 0.1562818943197445  
Especificidad: 0.9937183083159742

Métricas clase Calles  
Coeficiente DICE: 0.8697481722588042  
IOU: 0.7738997569978988  
Precisión: 0.8734017822656917  
Sensibilidad: 0.8741547101660384  
Especificidad: 0.968684175877266

Métricas clase Aceras  
Coeficiente DICE: 0.6852148994170504  
IOU: 0.5449078794342519  
Precisión: 0.7009631310758856  
Sensibilidad: 0.7014967964632842  
Especificidad: 0.9729283050644516

Métricas clase Vegetación  
Coeficiente DICE: 0.21276596378480736  
IOU: 0.13516311390424515  
Precisión: 0.2419388908880438  
Sensibilidad: 0.2636780604333115  
Especificidad: 0.9146580911252213

Métricas clase Vehículos  
Coeficiente DICE: 0.9406244157802439  
IOU: 0.8936232991570031  
Precisión: 0.9436444693739654  
Sensibilidad: 0.9476752311965161  
Especificidad: 0.9871345504362465

Métricas clase Paredes  
Coeficiente DICE: 0.07249086671064953  
IOU: 0.04592795082121143  
Precisión: 0.1433433573747113  
Sensibilidad: 0.13199943119193577  
Especificidad: 0.9918386224617797

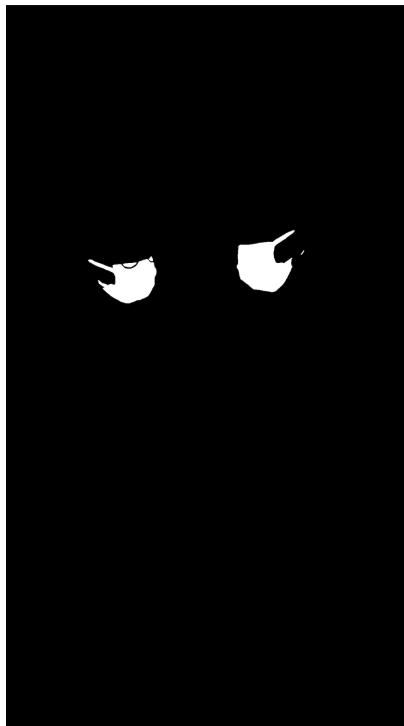
Métricas clase Señales  
Coeficiente DICE: 0.0  
IOU: 0.0  
Precisión: 0.0  
Sensibilidad: 0.0  
Especificidad: 0.9997434433353537

# Segmentación semántica en TF

## Segmentación semántica binaria

### Ejercicio 1

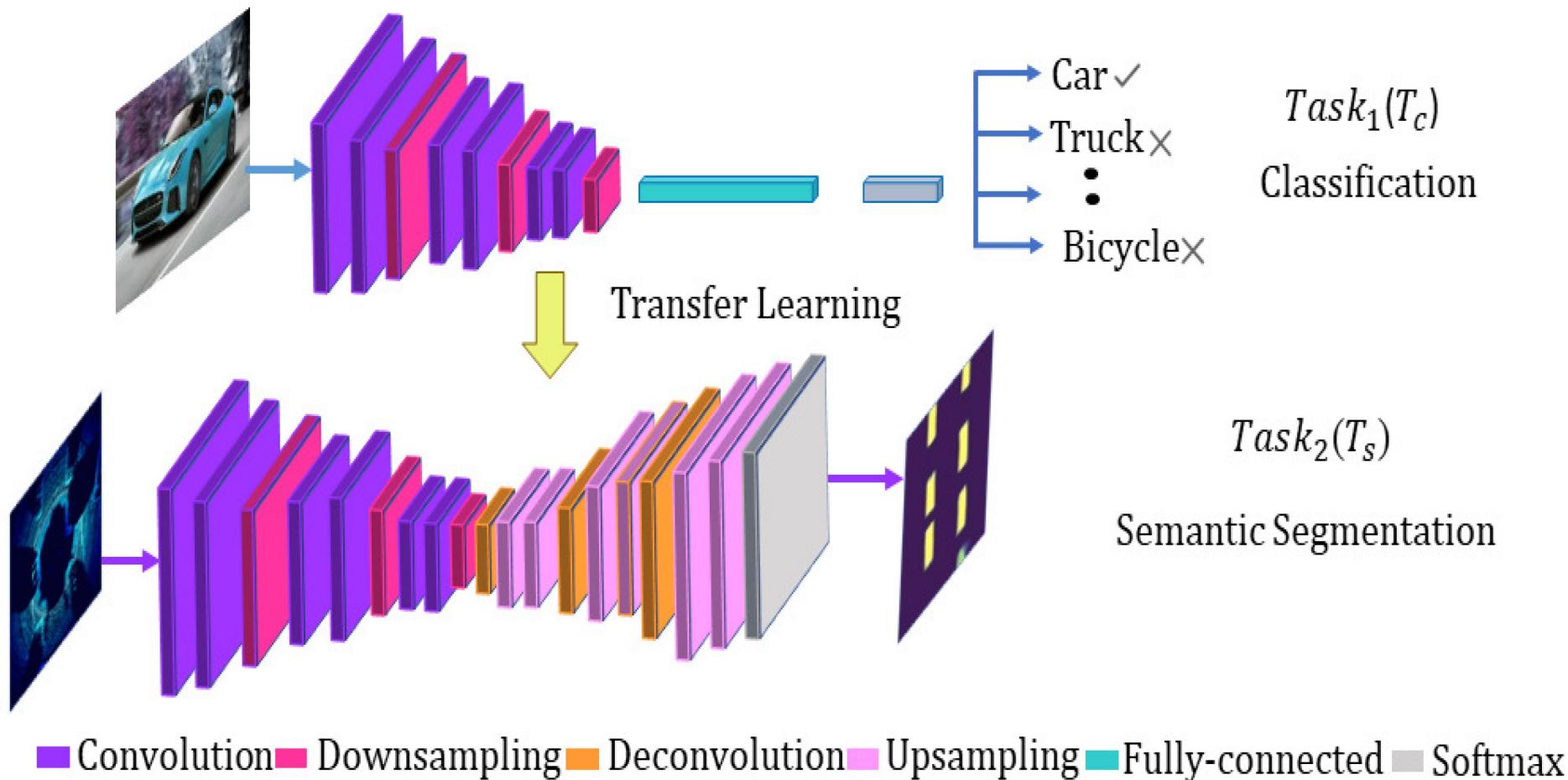
Ahora vais a resolver vosotros un problema de segmentación semántica. En este caso se trata de segmentación binaria. Por lo tanto, habrá que adaptar el código para poder resolver un problema de estas características



- Adapta el método de lectura de imágenes de forma que se generen matrices de segmentación con las dimensiones [N FILAS, N COLUMNAS, 1] donde esté a 1 los píxeles pertenecientes a la mascarilla. La imagen original se trata de una imagen RGB con valor (255,255,255) en los píxeles mascarilla.
- Genera dos datasets: entrenamiento y validación.
- Adapta la arquitectura.
- Adapta el entrenamiento.
- Extrae métricas sobre el conjunto de validación (como si fuera el de test).

# **Transferencia de conocimiento**

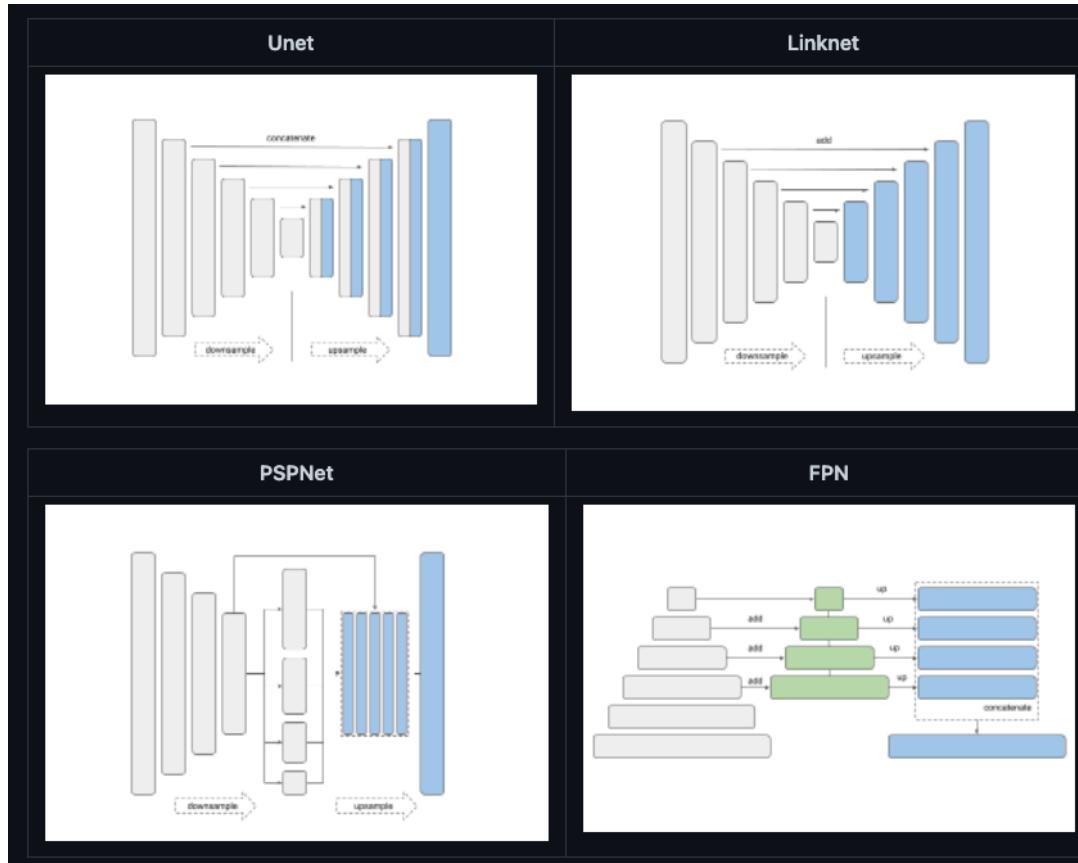
# Transferencia de conocimiento



# Transferencia de conocimiento

[https://github.com/qubvel/segmentation\\_models](https://github.com/qubvel/segmentation_models)

! pip install -U segmentation-models



## Backbones

Type	Names
VGG	'vgg16' 'vgg19'
ResNet	'resnet18' 'resnet34' 'resnet50' 'resnet101' 'resnet152'
SE-ResNet	'seresnet18' 'seresnet34' 'seresnet50' 'seresnet101' 'seresnet152'
ResNeXt	'resnext50' 'resnext101'
SE-ResNeXt	'seresnext50' 'seresnext101'
SENet154	'senet154'
DenseNet	'densenet121' 'densenet169' 'densenet201'
Inception	'inceptionv3' 'inceptionresnetv2'
MobileNet	'mobilenet' 'mobilenetv2'
EfficientNet	'efficientnetb0' 'efficientnetb1' 'efficientnetb2' 'efficientnetb3' 'efficientnetb4' 'efficientnetb5' 'efficientnetb6' 'efficientnetb7'

All backbones have weights trained on 2012 ILSVRC ImageNet dataset (`encoder_weights='imagenet'`).

# Transferencia de conocimiento

```
N_ROWS = 256
N_COLS = 256
N_LABELS = 13

def read_image(image_path, mask_path):
    # Leemos la imagen del directorio
    image = tf.io.read_file(image_path)
    image = tf.image.decode_png(image, channels=3)
    image = tf.image.convert_image_dtype(image, tf.float32)
    image = tf.image.resize(image, (N_ROWS, N_COLS), method='nearest')

    # Leemos la máscara del directorio
    mask = tf.io.read_file(mask_path)
    mask = tf.image.decode_png(mask, channels=3)
    mask = tf.math.reduce_max(mask, axis=-1, keepdims=True)
    mask = tf.image.resize(mask, (N_ROWS, N_COLS), method='nearest')
    mask = tf.squeeze(mask, axis=-1)
    mask = tf.one_hot(tf.cast(mask, tf.int32), N_LABELS)

    return image, mask
```

¿Qué deberíamos cambiar para adaptar la lectura de imágenes para emplear un modelo pre-entrenado?

# Transferencia de conocimiento

```
N_ROWS = 256
N_COLS = 256
N_LABELS = 13

BACKBONE = 'mobilenetv2'
preprocess_input = sm.get_preprocessing(BACKBONE)

def read_image(image_path, mask_path):
    # Leemos la imagen del directorio
    image = tf.io.read_file(image_path)
    image = tf.image.decode_png(image, channels=3)
    image = tf.cast(image, tf.float32)
    image = preprocess_input(image)
    image = tf.image.resize(image, (N_ROWS, N_COLS), method='nearest')

    # Leemos la máscara del directorio
    mask = tf.io.read_file(mask_path)
    mask = tf.image.decode_png(mask, channels=3)
    mask = tf.math.reduce_max(mask, axis=-1, keepdims=True)
    mask = tf.image.resize(mask, (N_ROWS, N_COLS), method='nearest')
    mask = tf.squeeze(mask, axis=-1)
    mask = tf.one_hot(tf.cast(mask, tf.int32), N_LABELS)

    return image, mask
```

# Transferencia de conocimiento

Importamos el modelo que queramos y que arquitectura queremos emplear en el codificador (BACKBONE), el número de clases, la función de activación y los pesos que queremos emplear de base.

```
model_t1_unet = sm.Unet(BACKBONE, input_shape=(N_ROWS, N_ROWS, 3), classes=N_LABELS,
                        activation='softmax', encoder_weights='imagenet', encoder_freeze=True)
```

El resto de pasos son exactamente igual, primero compilamos el modelo especificando el optimizador, la función de pérdidas y las métricas. Para las pérdidas y las métricas, podemos emplear las comunes, funciones propias o, la propia librería, nos ofrece unas cuantas opciones\*.

```
model_t1_unet.compile(optimizer='adam',
                      loss=sm.losses.DiceLoss(),
                      metrics=['accuracy', dice_coef_multiclass])
```

[https://github.com/qubvel/segmentation\\_models/blob/master/segmentation\\_models/losses.py](https://github.com/qubvel/segmentation_models/blob/master/segmentation_models/losses.py)

[https://github.com/qubvel/segmentation\\_models/blob/master/segmentation\\_models/metrics.py](https://github.com/qubvel/segmentation_models/blob/master/segmentation_models/metrics.py)

# Transferencia de conocimiento

```
EPOCHS = 100

path_experiment = path_models / 'Train4'
path_experiment.mkdir(exist_ok=True, parents = True)

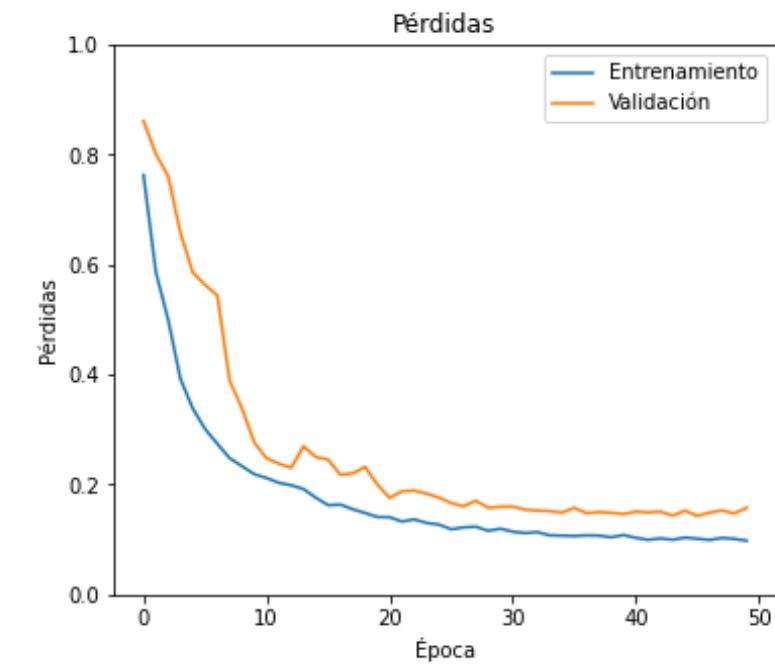
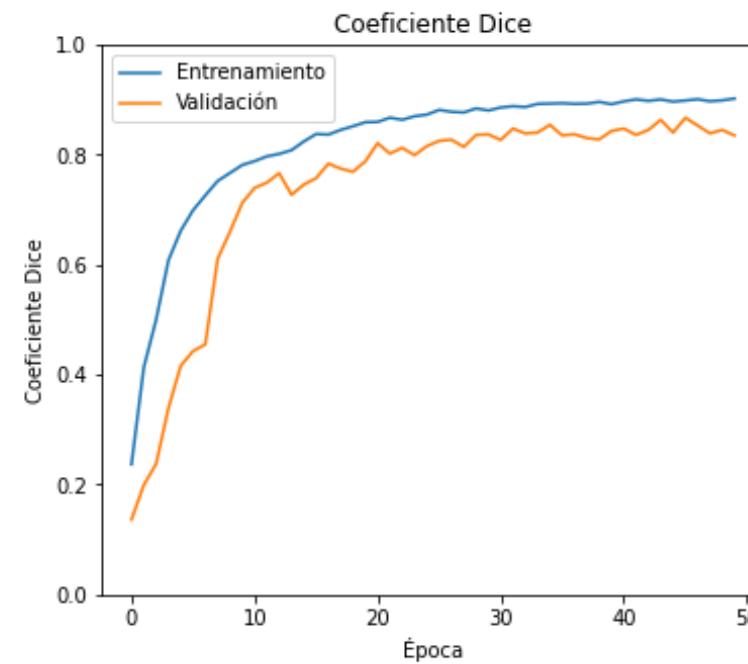
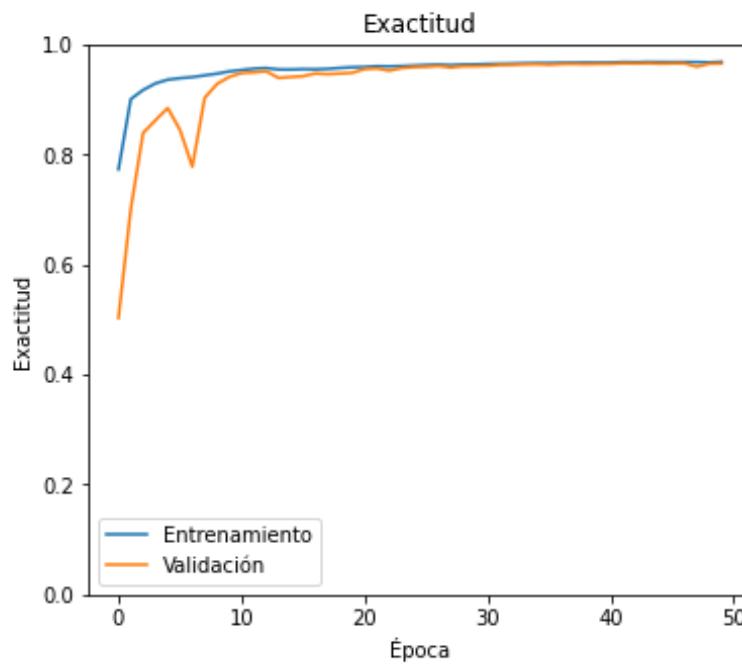
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=path_experiment / 'model.h5',
    monitor='val_loss',
    mode='min',
    save_best_only=True,
    verbose=1)

early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=5,
    verbose=1,
    mode='min',
    restore_best_weights=False
)

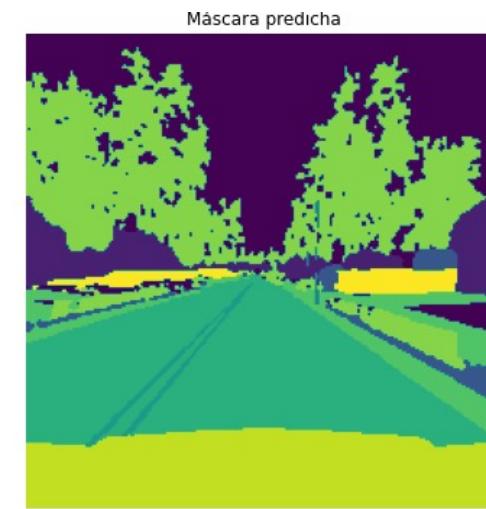
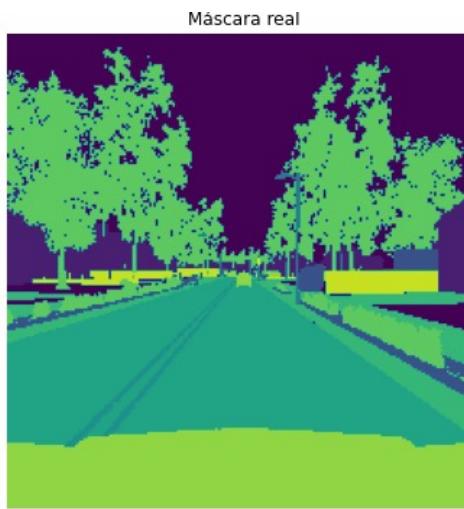
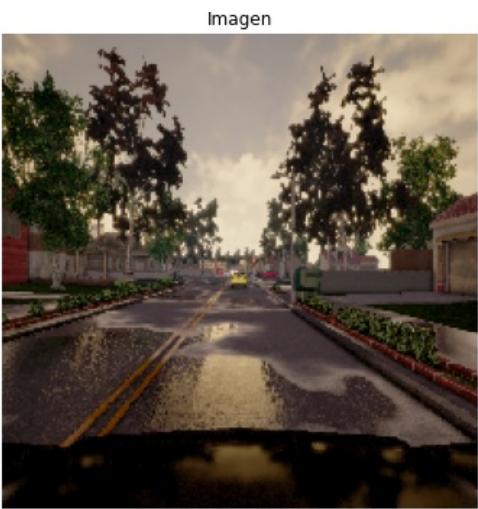
history_unet_tl = model_tl_unet.fit(train_dataset,
    epochs=EPOCHS,
    validation_data=validation_dataset,
    callbacks=[model_checkpoint_callback, early_stopping],
    verbose=1)

np.save(path_experiment / 'history.npy', history_unet_tl.history)
```

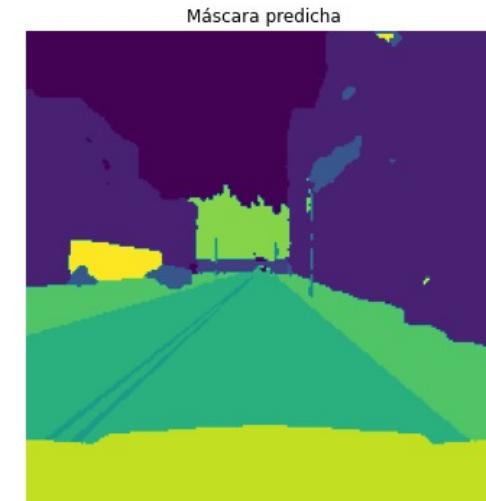
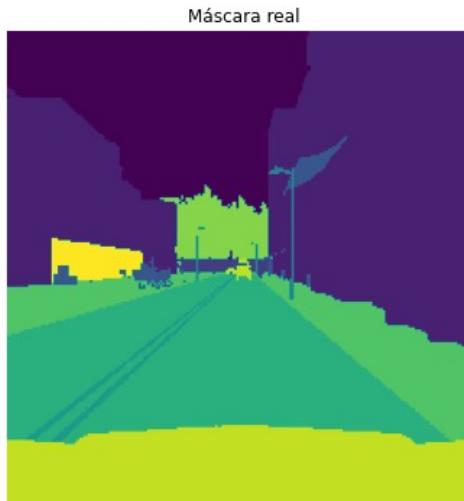
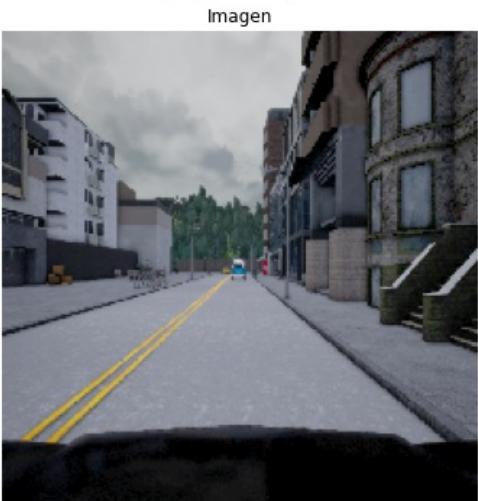
# Transferencia de conocimiento



# Transferencia de conocimiento

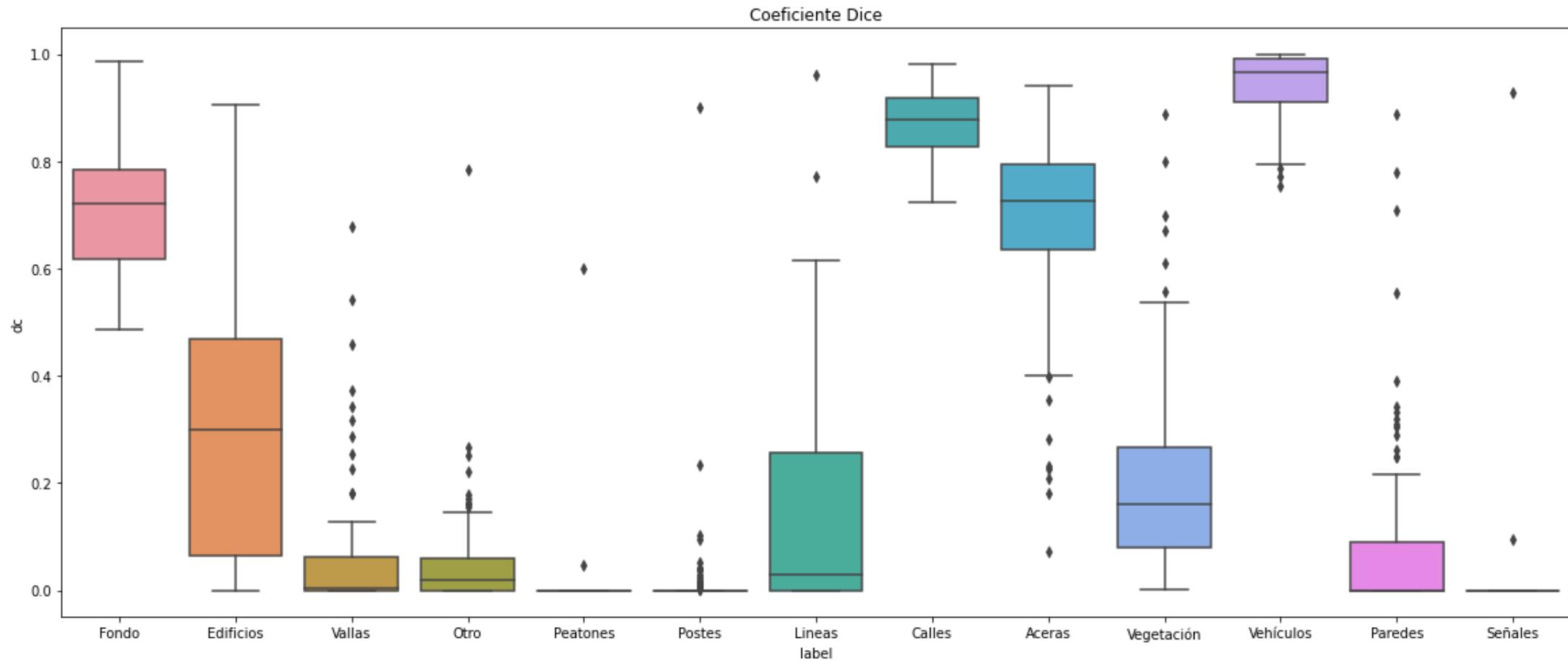


Etiquetas máscara real: [ 0 1 2 3 5 6 7 8 9 10 11 12]  
Etiquetas máscara predicha: [ 0 1 3 5 6 7 8 9 10 11]  
Clase Vallas (2) no segmentada.  
Clase Señales (12) no segmentada.



Etiquetas máscara real: [ 0 1 2 3 5 6 7 8 9 10 11]  
Etiquetas máscara predicha: [ 0 1 2 3 5 6 7 8 9 10 11]

# Transferencia de conocimiento



# Análisis de la base de datos

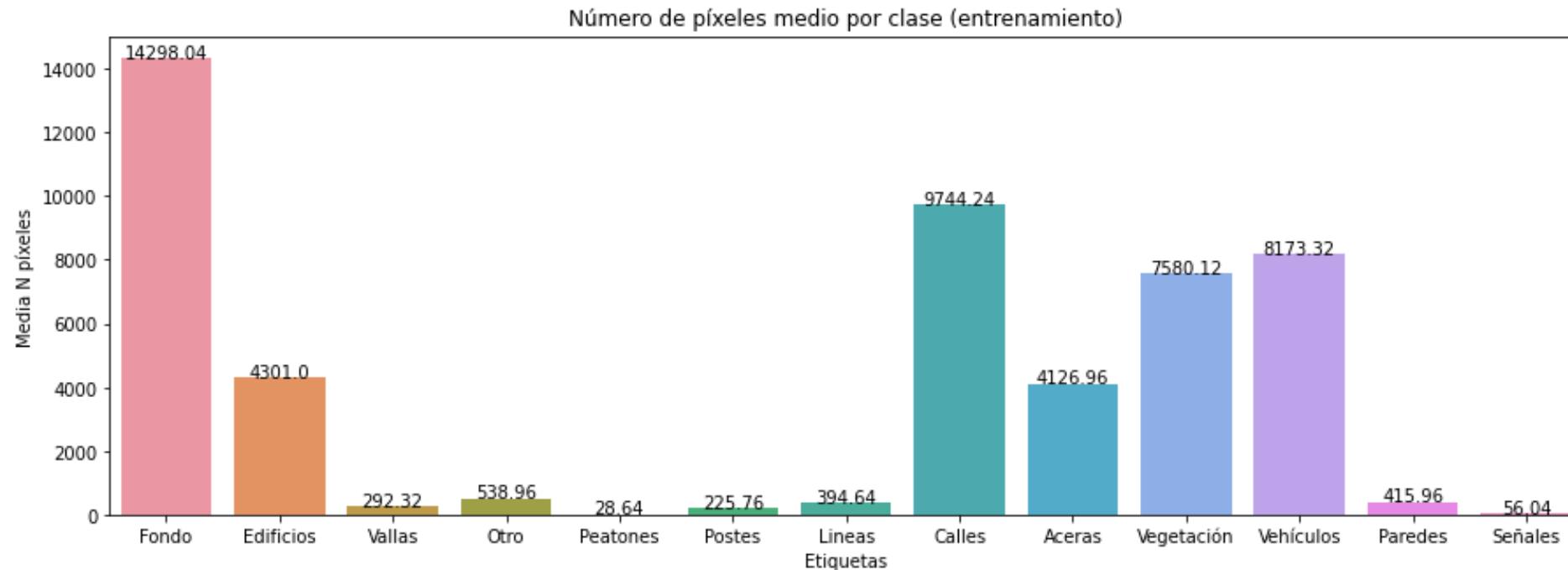
Vemos que, independientemente de la arquitectura y la aproximación empleada, las métricas para ciertas clases son muy malas.

Vamos a tratar de analizar un poco la base de datos de la que disponemos. Para ello, vamos a calcular el número de píxeles medio que hay para cada una de las clases en cada uno de los subconjuntos de datos.

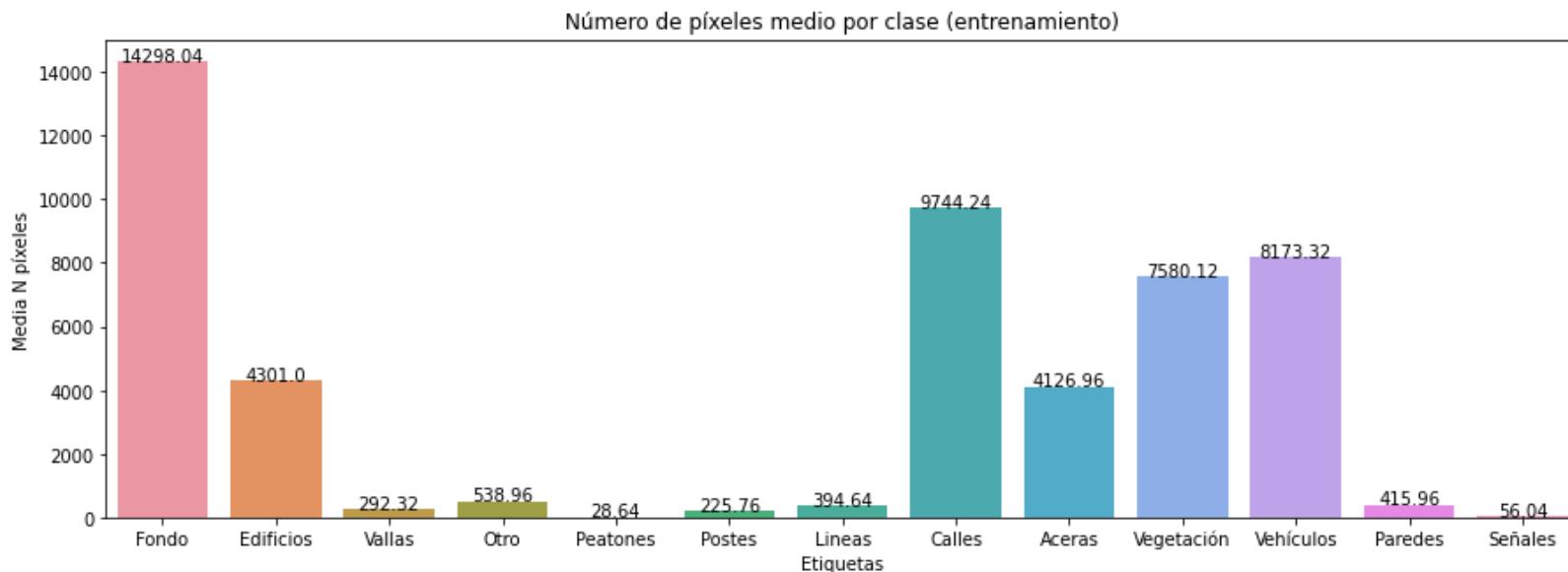
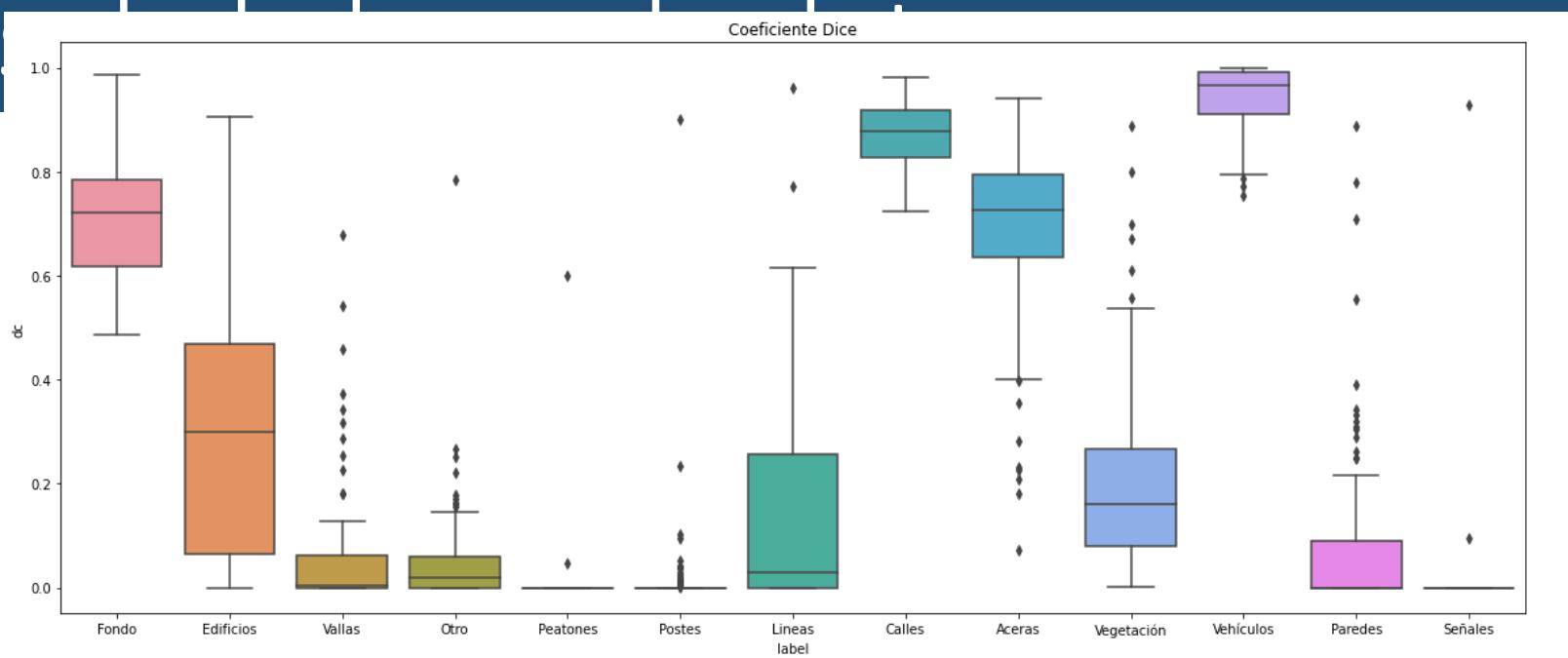
# Análisis de la base de datos

Vemos que, independientemente de la arquitectura y la aproximación empleada, las métricas para ciertas clases son muy malas.

Vamos a tratar de analizar un poco la base de datos de la que disponemos. Para ello, vamos a calcular el número de píxeles medio que hay para cada una de las clases en cada uno de los subconjuntos de datos.

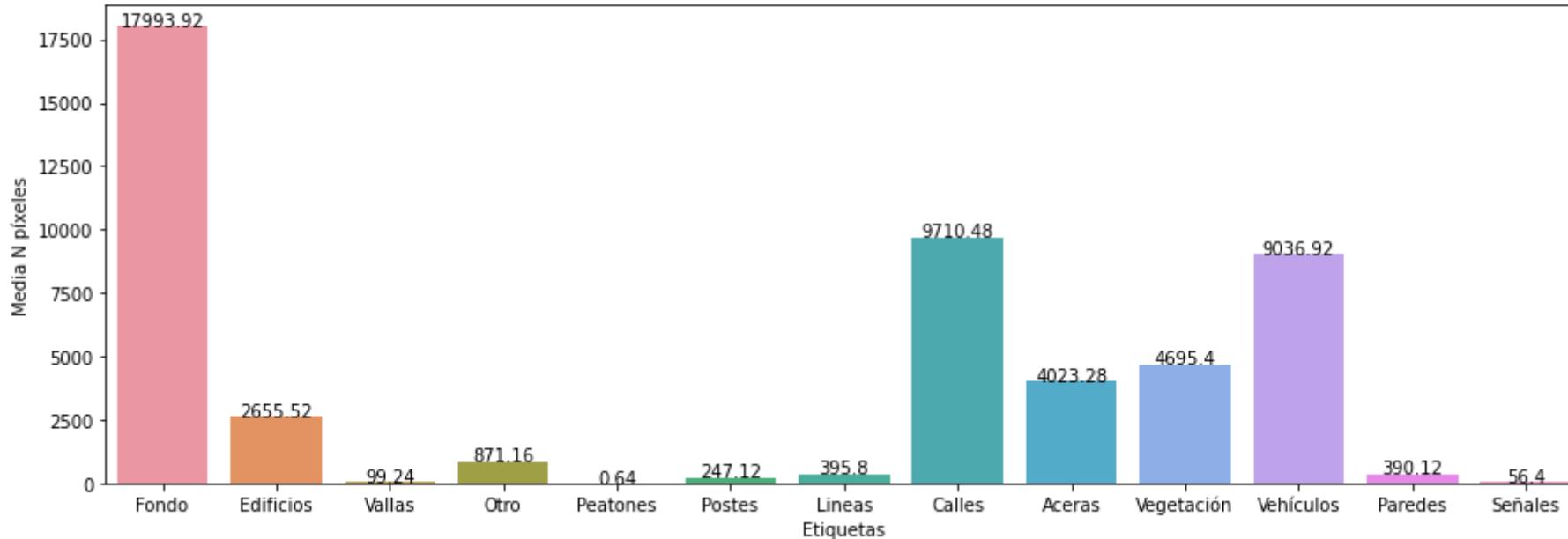


# Análisis

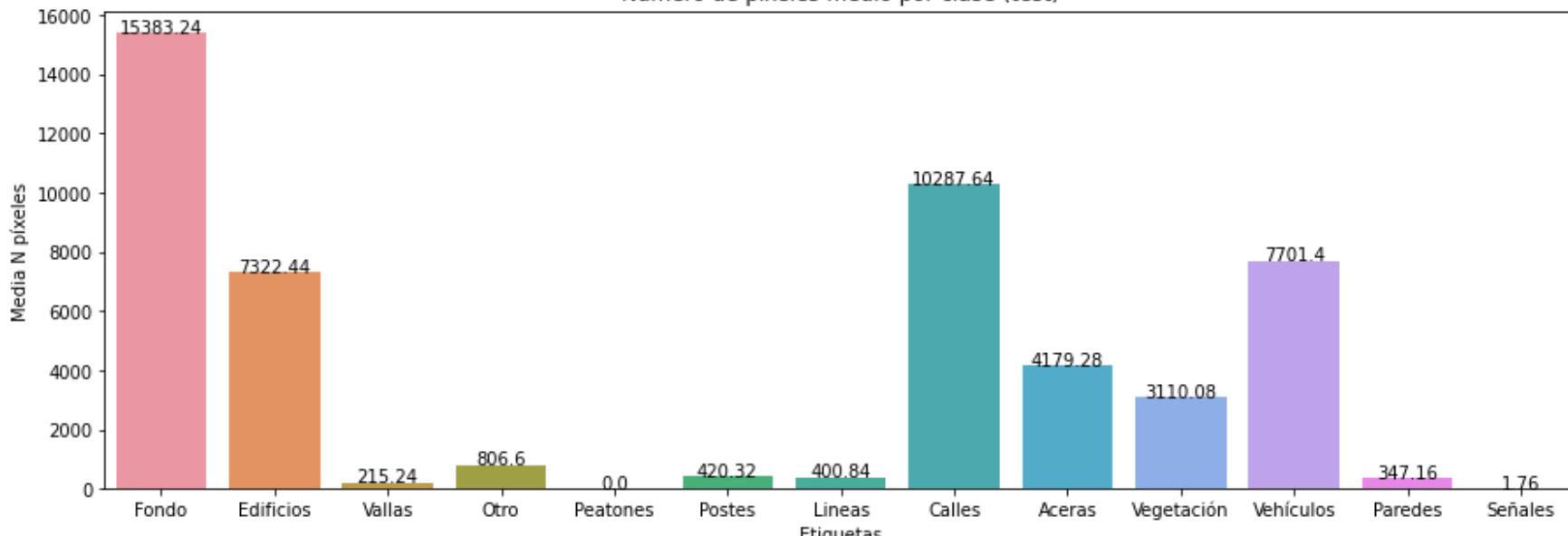


# Análi

Número de píxeles medio por clase (validación)



Número de píxeles medio por clase (test)



# Aumento de datos

# Aumento de datos

La principal peculiaridad del aumento de datos en un problema de segmentación es que las mismas transformaciones que le apliquemos a la imagen (relacionadas con alteración de la posición de los píxeles) se las deberemos aplicar a la máscara.

Para aplicar aumento de datos vamos a emplear las mismas librerías que vimos en el Tema 4: ***Albumentations*** para crear las transformaciones y ***ImageDataAugmentor*** para facilitar la generación de generadores de datos.

# Aumento de datos

## Cargado de datos

Cargado de datos que vimos en el Tema 5. Lee los archivos que encuentra directamente de la carpeta:

```
train_dataset = train_datagen.flow_from_directory(  
    data_root,  
    subset="training",  
    target_size=IMG_SIZE,  
    batch_size=BATCH_SIZE,  
    class_mode='sparse',  
    shuffle=True)
```



En este caso vamos a emplear otro método: “flow\_from\_dataframe”.

```
training_generator = train_datagen.flow_from_dataframe(train_df,  
    x_col='image_dirs', y_col='mask_dirs', class_mode='color_target',  
    batch_size=BATCH_SIZE, target_size=IMG_SIZE, shuffle=True)
```

**x\_col='image\_dirs', y\_col='mask\_dirs',** Especificamos directorios imágenes (x\_col) y etiquetas/máscaras (y\_col) con la información del DataFrame (train\_df)

**class\_mode='color\_target'**, Lee la etiqueta (target) como color (RGB)

# Aumento de datos

## Cargado de datos

```
def paths_to_dataframe(x_dir, y_dir):
    x_df = pd.DataFrame({'image_dirs':[str(f) for f in Path(x_dir).glob('*.*')]} )
    x_df['image_id'] = x_df.image_dirs.apply(lambda x: Path(x).stem)
    y_df = pd.DataFrame({'mask_dirs':[str(f) for f in Path(y_dir).glob('*.*')]} )
    y_df['image_id'] = y_df.mask_dirs.apply(lambda x: Path(x).stem)
    data_df = x_df.merge(y_df)
    return data_df
```

```
imgs_dir = '/content/dataset_segmentation/dataset_segmentation/CameraRGB'
masks_dir = '/content/dataset_segmentation/dataset_segmentation/CameraSeg'

train_df = paths_to_dataframe(os.path.join(imgs_dir, 'train'), os.path.join(masks_dir, 'train'))
valid_df = paths_to_dataframe(os.path.join(imgs_dir, 'val'), os.path.join(masks_dir, 'val'))
```

1 to 5

index	image_fn	image_id	mask_fn
0	/content/dataset_segmentation/dataset_segmentation/CameraRGB/train/F64-99.png	F64-99	/content/dataset_segmentation/dataset_segmentation/CameraSeg/train/F64-99.png
1	/content/dataset_segmentation/dataset_segmentation/CameraRGB/train/F61-52.png	F61-52	/content/dataset_segmentation/dataset_segmentation/CameraSeg/train/F61-52.png
2	/content/dataset_segmentation/dataset_segmentation/CameraRGB/train/F67-4.png	F67-4	/content/dataset_segmentation/dataset_segmentation/CameraSeg/train/F67-4.png
3	/content/dataset_segmentation/dataset_segmentation/CameraRGB/train/F66-48.png	F66-48	/content/dataset_segmentation/dataset_segmentation/CameraSeg/train/F66-48.png
4	/content/dataset_segmentation/dataset_segmentation/CameraRGB/train/F61-15.png	F61-15	/content/dataset_segmentation/dataset_segmentation/CameraSeg/train/F61-15.png

# Aumento de datos

## Cargado de datos

```
AUGMENTATIONS = A.Compose([
    A.Transpose(p=0.5),
    A.Flip(p=0.5),
    A.OneOf([
        A.RandomBrightnessContrast(brightness_limit=0.3, contrast_limit=0.3),
        A.RandomBrightnessContrast(brightness_limit=0.1, contrast_limit=0.1)
    ],p=1),
    A.GaussianBlur(p=0.05),
    A.HueSaturationValue(p=0.5),
])
```

# Aumento de datos

## Cargado de datos

```
SEED = 123

# Generador de datos de entrenamiento
train_datagen = ImageDataAugmentor(
    augment=AUGMENTATIONS,
    label_augment_mode="mask",
    preprocess_input=preprocess_input,
    preprocess_labels=one_hot_encode_masks,
    validation_split=0.2,
    seed=SEED
)

# Generador de datos de validación --> !!! No
# aplicamos aumento de datos !!!
val_datagen = ImageDataAugmentor(
    preprocess_input=preprocess_input,
    preprocess_labels=one_hot_encode_masks,
    validation_split=0.2,
    seed=SEED
)
```

```
from keras.utils.np_utils import to_categorical

N_LABELS = 13

def one_hot_encode_masks(mask):
    mask = np.max(mask, axis=-1)
    mask = to_categorical(mask, num_classes = N_LABELS)
    return mask

def preprocess_input(image):
    image = image.astype('float32')
    image /= 255

    return image
```

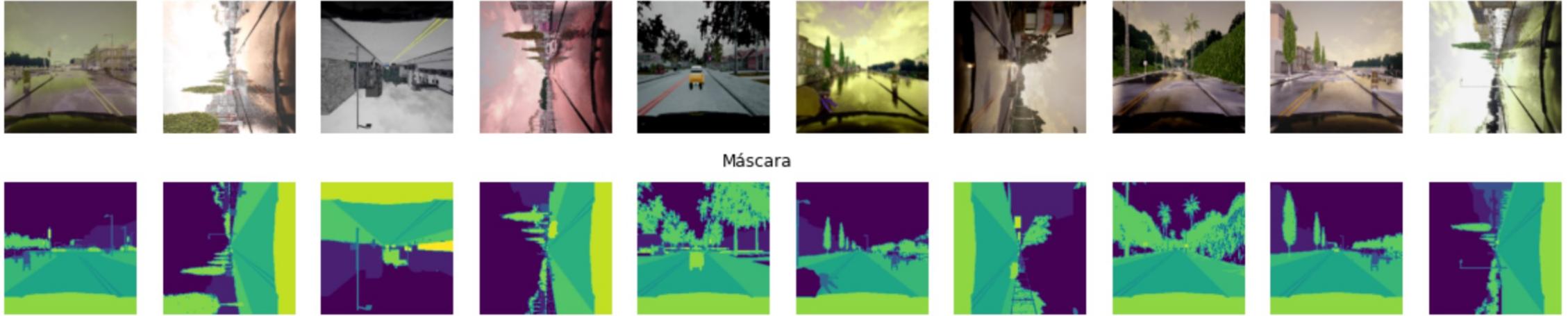
# Aumento de datos

## Cargado de datos

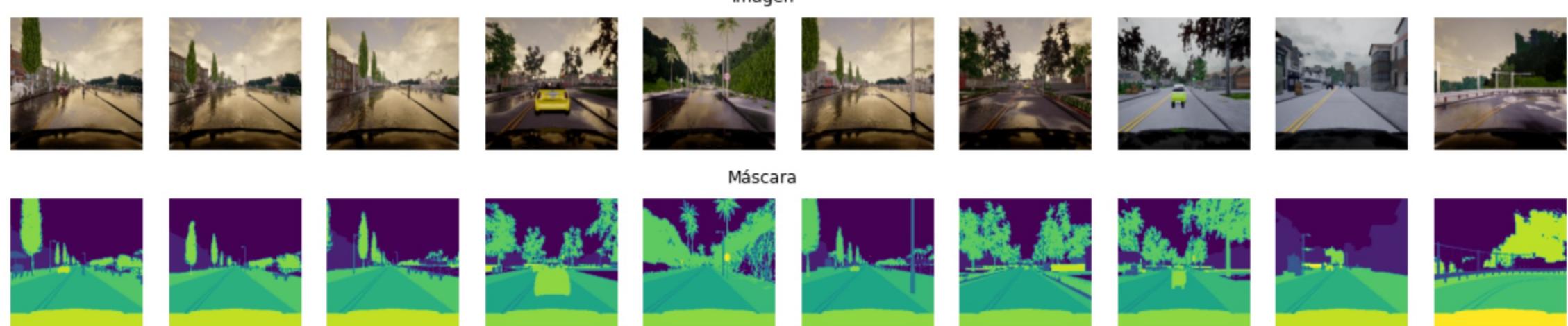
```
training_generator = train_datagen.flow_from_dataframe(  
    train_df,  
    x_col='image_dirs',  
    y_col='mask_dirs',  
    class_mode='color_target', #<- lee la etiqueta (targets) de `mask_dirs` como color (RGB)  
    batch_size=BATCH_SIZE,  
    target_size=(N_ROWS, N_COLS),  
    shuffle=True  
)  
  
validation_generator = val_datagen.flow_from_dataframe(  
    valid_df,  
    x_col='image_dirs',  
    y_col='mask_dirs',  
    class_mode='color_target',  
    batch_size=BATCH_SIZE,  
    target_size=(N_ROWS, N_COLS),  
    shuffle=True  
)
```

# Aumento de datos

Entrenamiento



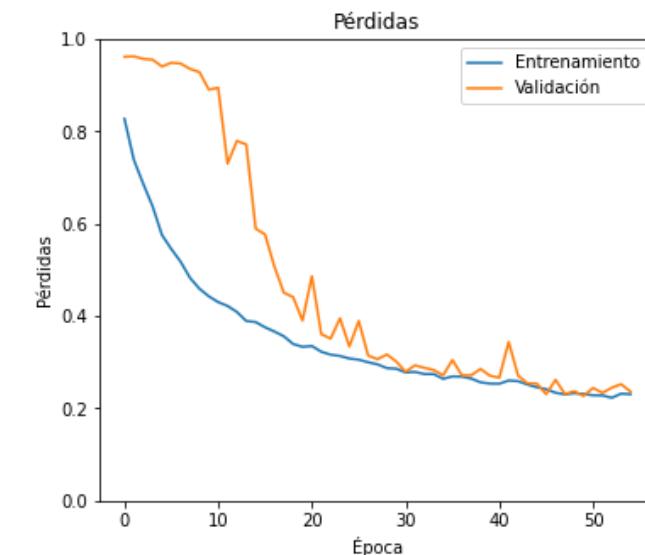
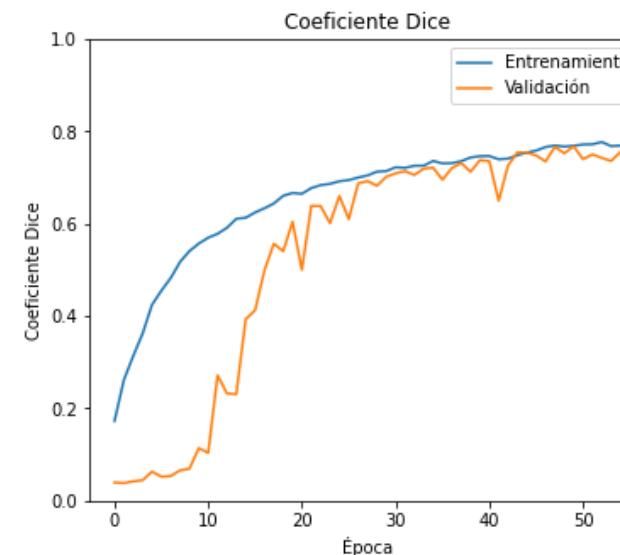
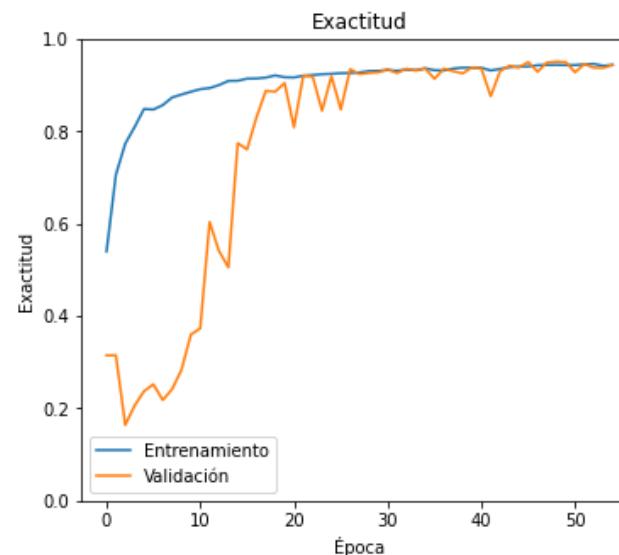
Validación



# Aumento de datos

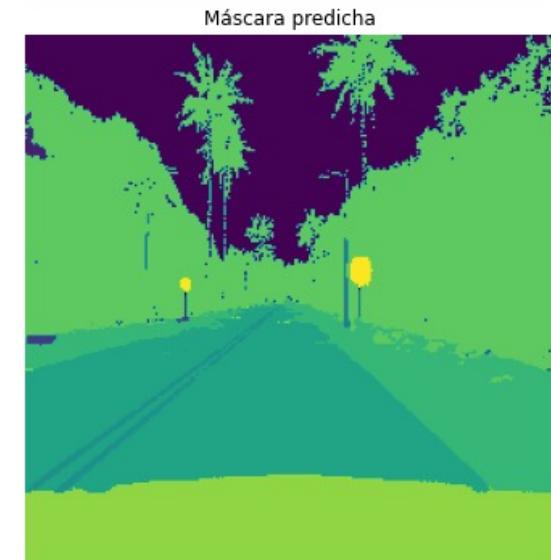
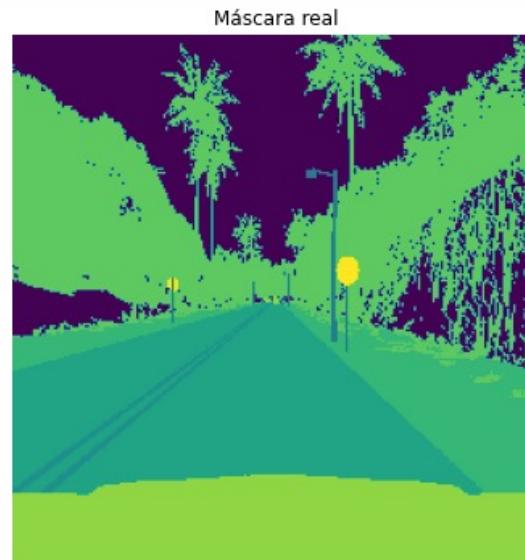
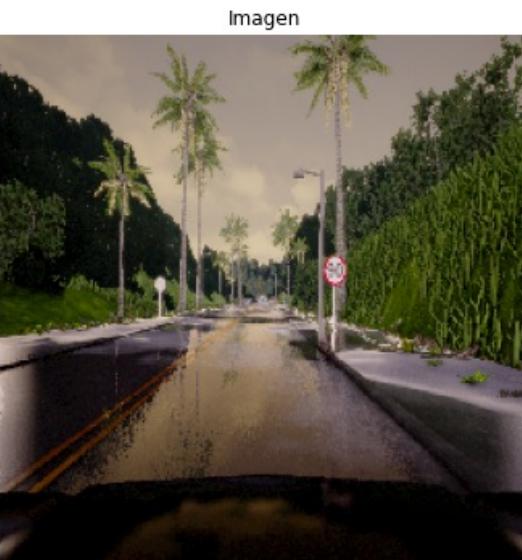
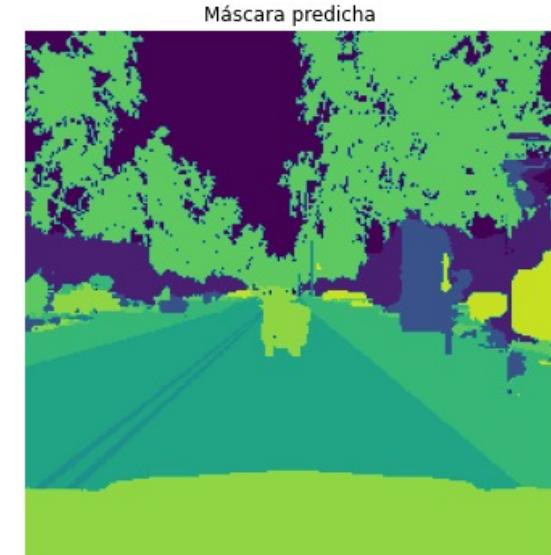
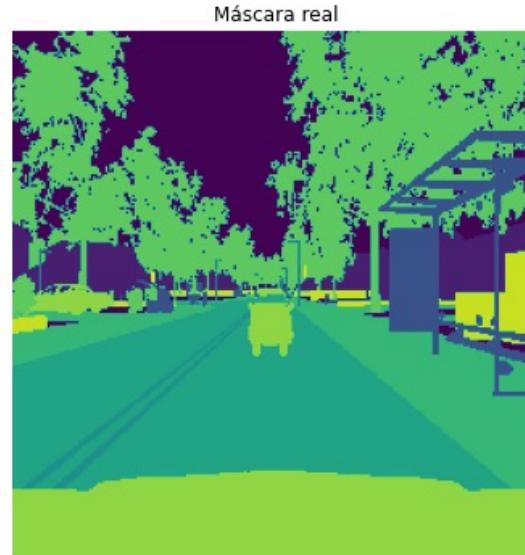
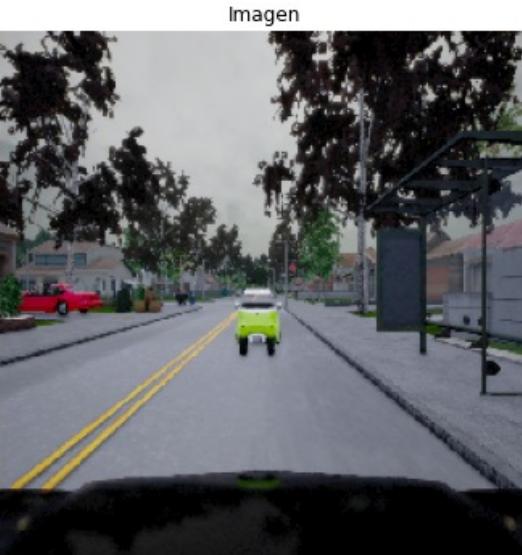
## Entrenamiento

```
history_unet = model_unet.fit(  
    training_generator,  
    epochs=EPOCHS,  
    validation_data=validation_generator,  
    callbacks=[model_checkpoint_callback, early_stopping],  
    verbose=1)
```



# Aumento de datos

Test



# Tema 6. Segmentación

Ana Jiménez Pastor  
[anjipas@gmail.com](mailto:anjipas@gmail.com)