```
 import os
        import numpy as np
        import tensorflow as tf
        from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, 
UpSampling2D, Concatenate, Dropout
        from tensorflow.keras.models import Model
        from tensorflow.keras.optimizers import Adam
        from tensorflow.keras.metrics import MeanIoU, Precision, Recall
        from tensorflow.keras.callbacks import ModelCheckpoint, 
EarlyStopping
        from tensorflow.keras.regularizers import l2
        from tensorflow.keras.losses import CategoricalCrossentropy
        import matplotlib.pyplot as plt

        # Directorios con imágenes y máscaras ya separadas
        train_image_dir = r"C:\Users\ismae\Desktop\TFG\Python 
Windows\frames"
        train_mask_dir = 
r"C:\Users\ismae\Desktop\TFG\Mascaras\entrenamiento"
        val_image_dir = r"C:\Users\ismae\Desktop\TFG\Python 
Windows\frames"
        val_mask_dir = r"C:\Users\ismae\Desktop\TFG\Mascaras\validacion"

        # Parámetros del modelo
        IMAGE_SIZE = (256, 256)  # Tamaño al que redimensionaremos las 
imágenes
        NUM_CLASSES = 4  # Número de clases (Fondo, hueso, tendón, 
límites)
        BATCH_SIZE = 4
        EPOCHS = 50

        def desescalar_masks(masks):
            return (masks / 85).astype(np.uint8)

        # Función para cargar imágenes y máscaras
        def load_images_and_masks(image_dir, mask_dir, image_size):
            images = []
            masks = []
            for file_name in os.listdir(image_dir):
                image_path = os.path.join(image_dir, file_name)
                mask_path = os.path.join(mask_dir, 
f"{os.path.splitext(file_name)[0]}_mask.png")

                if os.path.exists(image_path) and 
os.path.exists(mask_path):
                    # Cargar y redimensionar la imagen
                    image = 
tf.image.decode_image(tf.io.read_file(image_path), channels=3)
                    image = tf.image.resize(image, image_size)

                    # Cargar y redimensionar la máscara
                    mask = 
tf.image.decode_image(tf.io.read_file(mask_path), channels=1)
                    mask = tf.image.resize(mask, image_size, 
method='nearest')  # Mantener valores discretos
```

```
                    # Desescalar las máscaras (85, 170, 255 → 1, 2, 3)
                    mask = desescalar_masks(mask.numpy())

                    # Agregar a las listas
                    images.append(image.numpy())
                    masks.append(mask)

            images = np.array(images) / 255.0  # Normalizar imágenes (0-
1)
            masks = np.array(masks)  # Asegurar que las máscaras sean
enteros
            return images, masks

        # Cargar los datos
        X_train, y_train = load_images_and_masks(train_image_dir,
train_mask_dir, IMAGE_SIZE)
        X_val, y_val = load_images_and_masks(val_image_dir, val_mask_dir,
IMAGE_SIZE)

        # Convertir máscaras a categóricas (One-Hot Encoding)
        y_train = tf.keras.utils.to_categorical(y_train,
num_classes=NUM_CLASSES)
        y_val = tf.keras.utils.to_categorical(y_val,
num_classes=NUM_CLASSES)

        print(f"Imágenes de entrenamiento: {X_train.shape}, Máscaras:
{y_train.shape}")
        print(f"Imágenes de validación: {X_val.shape}, Máscaras:
{y_val.shape}")

        def weighted_loss(y_true, y_pred):
            class_weights = tf.constant([0.2, 22.98, 26.25, 25.42],
dtype=tf.float32)  # Pesos de ejemplo
            y_true = tf.cast(tf.argmax(y_true, axis=-1), tf.int32)
            sample_weights = tf.gather(class_weights, y_true)
            cce = CategoricalCrossentropy(from_logits=False)
            loss = cce(y_true=tf.one_hot(y_true, depth=4), y_pred=y_pred)
            return loss * sample_weights

        # Definir el modelo U-Net
        def unet_model(input_size=(256, 256, 3), num_classes=4):
            inputs = Input(input_size)
            s = tf.keras.layers.Lambda(lambda x: x / 255.0)(inputs)  #
Normalización integrada

            # Encoder
            c1 = Conv2D(64, (3, 3), activation='relu',
kernel_initializer='he_normal',
                        padding='same', kernel_regularizer=l2(0.001))(s)
            c1 = Dropout(0.4)(c1)
            c1 = Conv2D(64, (3, 3), activation='relu',
kernel_initializer='he_normal',
                        padding='same', kernel_regularizer=l2(0.001))(c1)
            p1 = MaxPooling2D((2, 2))(c1)
```

```
            c2 = Conv2D(128, (3, 3), activation='relu',
kernel_initializer='he_normal',
                        padding='same', kernel_regularizer=l2(0.001))(p1)
            c2 = Dropout(0.4)(c2)
            c2 = Conv2D(128, (3, 3), activation='relu',
kernel_initializer='he_normal',
                        padding='same', kernel_regularizer=l2(0.001))(c2)
            p2 = MaxPooling2D((2, 2))(c2)

            # Bottleneck
            c3 = Conv2D(256, (3, 3), activation='relu',
kernel_initializer='he_normal',
                        padding='same', kernel_regularizer=l2(0.001))(p2)
            c3 = Dropout(0.4)(c3)
            c3 = Conv2D(256, (3, 3), activation='relu',
kernel_initializer='he_normal',
                        padding='same', kernel_regularizer=l2(0.001))(c3)

            # Decoder
            u1 = UpSampling2D((2, 2))(c3)
            u1 = Concatenate()([u1, c2])
            c4 = Conv2D(128, (3, 3), activation='relu',
kernel_initializer='he_normal',
                        padding='same')(u1)
            c4 = Dropout(0.4)(c4)
            c4 = Conv2D(128, (3, 3), activation='relu',
kernel_initializer='he_normal',
                        padding='same')(c4)

            u2 = UpSampling2D((2, 2))(c4)
            u2 = Concatenate()([u2, c1])
            c5 = Conv2D(64, (3, 3), activation='relu',
kernel_initializer='he_normal',
                        padding='same')(u2)
            c5 = Dropout(0.4)(c5)
            c5 = Conv2D(64, (3, 3), activation='relu',
kernel_initializer='he_normal',
                        padding='same')(c5)

            outputs = Conv2D(num_classes, (1, 1),
activation='softmax')(c5)

            model = Model(inputs, outputs)
            return model

        # Crear el modelo
        model = unet_model(input_size=(IMAGE_SIZE[0], IMAGE_SIZE[1], 3),
num_classes=NUM_CLASSES)
        model.summary()

        # Compilar el modelo
        model.compile(optimizer=Adam(learning_rate=1e-4),
                    loss=weighted_loss,
                    metrics=['accuracy',
tf.keras.metrics.MeanIoU(num_classes=NUM_CLASSES), Precision(),
Recall()])
```

```
        # Configurar el callback para guardar el modelo con los mejores
pesos
        checkpoint = ModelCheckpoint(
            filepath=r"C:\Users\ismae\Desktop\TFG\model_unet_best3_1.h5",
  # Guarda el modelo completo con los mejores pesos
            monitor='val_loss',  # Métrica para determinar los mejores
pesos
            save_best_only=True,  # Solo guardar cuando los pesos son
mejores
            save_weights_only=False,  # Guardar el modelo completo
            mode='min',  # Buscar la pérdida mínima
            verbose=1
        )
        early_stopping = EarlyStopping(monitor='val_loss', patience=10,
restore_best_weights=True)

        # Entrenar el modelo
        history = model.fit(
            X_train, y_train,
            validation_data=(X_val, y_val),
            batch_size=BATCH_SIZE,
            epochs=EPOCHS,
            verbose=1,
            callbacks=[checkpoint, early_stopping]  # Callback para
guardar los mejores pesos
        )

        # Guardar el modelo completo al final del entrenamiento
        model.save(r"C:\Users\ismae\Desktop\TFG\model_unet_final3_1.h5")

        # Graficar la historia de entrenamiento
        plt.figure(figsize=(12, 6))
        plt.plot(history.history['accuracy'], label='Precisión de
entrenamiento')
        plt.plot(history.history['val_accuracy'], label='Precisión de
validación')
        plt.plot(history.history['loss'], label='Pérdida de
entrenamiento')
        plt.plot(history.history['val_loss'], label='Pérdida de
validación')
        plt.legend()
        plt.title("Historia de entrenamiento")
        plt.show()
```