```python
import os
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D,
UpSampling2D, Conv2DTranspose, 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,
ReduceLROnPlateau
from tensorflow.keras.regularizers import l2
from tensorflow.keras.losses import CategoricalCrossentropy
import matplotlib.pyplot as plt
import pandas as pd
import cv2


# Directorios con imágenes y máscaras ya separadas
train_image_dir = "/home/tfd/ihernand/prueba/TFG/frames"
train_mask_dir =
"/home/tfd/ihernand/prueba/TFG/frames/Mascaras/entrenamiento"
val_image_dir = "/home/tfd/ihernand/prueba/TFG/frames"
val_mask_dir = "/home/tfd/ihernand/prueba/TFG/frames/Mascaras/validacion"

# Crear directorio de salida
output_dir = "output_results"
os.makedirs(output_dir, exist_ok=True)

# Parámetros del modelo
IMAGE_SIZE = (128, 128)  # Tamaño al que redimensionaremos las imágenes
NUM_CLASSES = 4  # Número de clases (Fondo, hueso, tendón, límites)
BATCH_SIZE = 20
EPOCHS = 50

# Learning rate inicial
INITIAL_LR = 1e-3

# Pesos de clase ajustados manualmente
pesos_clase = [0.2, 22.98, 43.0, 55.0]
print(f"Pesos de clase manuales: {pesos_clase}")

# Función de pérdida con pesos manuales
def weighted_loss_manual(y_true, y_pred):
    class_weights = tf.constant(pesos_clase, dtype=tf.float32)
    y_true_indices = tf.cast(tf.argmax(y_true, axis=-1), tf.int32)
    sample_weights = tf.gather(class_weights, y_true_indices)
    y_true_one_hot = tf.one_hot(y_true_indices, depth=NUM_CLASSES)
    cce = CategoricalCrossentropy(from_logits=False)
    loss = cce(y_true=y_true_one_hot, y_pred=y_pred)
    weighted_loss = tf.reduce_mean(loss * sample_weights)
    return weighted_loss

# Función para cargar imágenes y máscaras
def desescalar_masks(masks):
    return (masks / 85).astype(np.uint8)

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_one_hot = tf.keras.utils.to_categorical(y_train,
num_classes=NUM_CLASSES)
y_val_one_hot = tf.keras.utils.to_categorical(y_val,
num_classes=NUM_CLASSES)
```

```
print(f"Imágenes de entrenamiento: {X_train.shape}, Máscaras:
{y_train_one_hot.shape}")
print(f"Imágenes de validación: {X_val.shape}, Máscaras:
{y_val_one_hot.shape}")

# Definir la arquitectura U-Net de tu compañero
def unet_companero(input_size=(128, 128, 3), num_classes=4):
    inputs = Input(input_size)
    s = tf.keras.layers.Lambda(lambda x: x / 255.0)(inputs)

    # Encoder
    c1 = Conv2D(16, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(s)
    c1 = Dropout(0.3)(c1)
    c1 = Conv2D(16, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(c1)
    p1 = MaxPooling2D((2, 2))(c1)

    c2 = Conv2D(32, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(p1)
    c2 = Dropout(0.3)(c2)
    c2 = Conv2D(32, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(c2)
    p2 = MaxPooling2D((2, 2))(c2)

    c3 = Conv2D(64, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(p2)
    c3 = Dropout(0.3)(c3)
    c3 = Conv2D(64, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(c3)
    p3 = MaxPooling2D((2, 2))(c3)

    c4 = Conv2D(128, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(p3)
    c4 = Dropout(0.3)(c4)
    c4 = Conv2D(128, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(c4)
    p4 = MaxPooling2D((2, 2))(c4)

    c5 = Conv2D(256, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(p4)
    c5 = Dropout(0.3)(c5)
    c5 = Conv2D(256, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(c5)

    # Decoder
    u6 = Conv2DTranspose(256, (2, 2), strides=(2, 2), padding='same')(c5)
    u6 = Concatenate()([u6, c4])
    c6 = Conv2D(256, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(u6)
    c6 = Dropout(0.3)(c6)
    c6 = Conv2D(256, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(c6)

    u7 = Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='same')(c6)
    u7 = Concatenate()([u7, c3])
    c7 = Conv2D(128, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(u7)
    c7 = Dropout(0.3)(c7)
    c7 = Conv2D(128, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(c7)

    u8 = Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same')(c7)
    u8 = Concatenate()([u8, c2])
    c8 = Conv2D(64, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(u8)
    c8 = Dropout(0.3)(c8)
    c8 = Conv2D(64, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(c8)

    u9 = Conv2DTranspose(32, (2, 2), strides=(2, 2), padding='same')(c8)
    u9 = Concatenate()([u9, c1])
    c9 = Conv2D(32, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(u9)
    c9 = Dropout(0.3)(c9)
    c9 = Conv2D(32, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(c9)

    outputs = Conv2D(num_classes, (1, 1), activation='softmax')(c9)

    model = Model(inputs, outputs)
    return model

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

# Configurar el callback para reducir el learning rate
reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.5,
    patience=5,
    min_lr=1e-6,
    verbose=1)

# Compilar el modelo
```

```
model.compile(optimizer=Adam(learning_rate=INITIAL_LR),
              loss=weighted_loss_manual,
              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=os.path.join(output_dir, "model_unet_best.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_one_hot,
    validation_data=(X_val, y_val_one_hot),
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    verbose=1,
    callbacks=[checkpoint, early_stopping, reduce_lr]  # Callback para
guardar los mejores pesos y reducir LR
)

# Guardar el modelo completo al final del entrenamiento
model.save(os.path.join(output_dir, "model_unet_final.h5"))

# Guardar métricas en un archivo CSV
metrics_df = pd.DataFrame(history.history)
metrics_df.to_csv(os.path.join(output_dir, "training_metrics.csv"),
index=False)

# 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.savefig(os.path.join(output_dir, "training_history.png"))
plt.show()
```

```
# Graficar precisión, recall y MeanIoU
plt.figure(figsize=(18, 6))
plt.subplot(1, 3, 1)
plt.plot(history.history['precision'], label='Precisión')
plt.plot(history.history['val_precision'], label='Val. Precisión')
plt.legend()
plt.title("Precisión")

plt.subplot(1, 3, 2)
plt.plot(history.history['recall'], label='Recall')
plt.plot(history.history['val_recall'], label='Val. Recall')
plt.legend()
plt.title("Recall")

plt.subplot(1, 3, 3)
plt.plot(history.history['mean_io_u'], label='Mean IoU')
plt.plot(history.history['val_mean_io_u'], label='Val. Mean IoU')
plt.legend()
plt.title("Mean IoU")

plt.tight_layout()
plt.savefig(os.path.join(output_dir, "training_metrics_plot.png"))
plt.show()


# --- Nuevo bloque para probar el modelo ---
def test_model_on_samples(test_image_dir, output_dir, model,
image_size=(128, 128), num_samples=5):
    os.makedirs(output_dir, exist_ok=True)
    test_images = sorted(os.listdir(test_image_dir))[:num_samples]

    for idx, image_file in enumerate(test_images):
        image_path = os.path.join(test_image_dir, image_file)
        image = cv2.imread(image_path)
        image_resized = cv2.resize(image, image_size) / 255.0
        input_image = np.expand_dims(image_resized, axis=0)

        predicted_mask = model.predict(input_image)
        predicted_mask = np.argmax(predicted_mask, axis=-1)[0]

        unique_classes = np.unique(predicted_mask)
        print(f"Imagen {image_file}: Clases únicas predichas -
{unique_classes}")

        plt.figure(figsize=(12, 4))
        plt.subplot(1, 2, 1)
        plt.imshow(image)
        plt.title("Imagen Original")
        plt.axis('off')

        plt.subplot(1, 2, 2)
        plt.imshow(predicted_mask, cmap='jet')
        plt.title("Máscara Predicha")
        plt.axis('off')

        plt.tight_layout()
        output_path = os.path.join(output_dir, f"predicted_{idx +
1}.png")
        plt.savefig(output_path)
        plt.close()

# Llamada a la función de prueba
test_model_on_samples("/home/tfd/ihernand/prueba/TFG/frames",
os.path.join(output_dir, "test_predictions"), model)
```