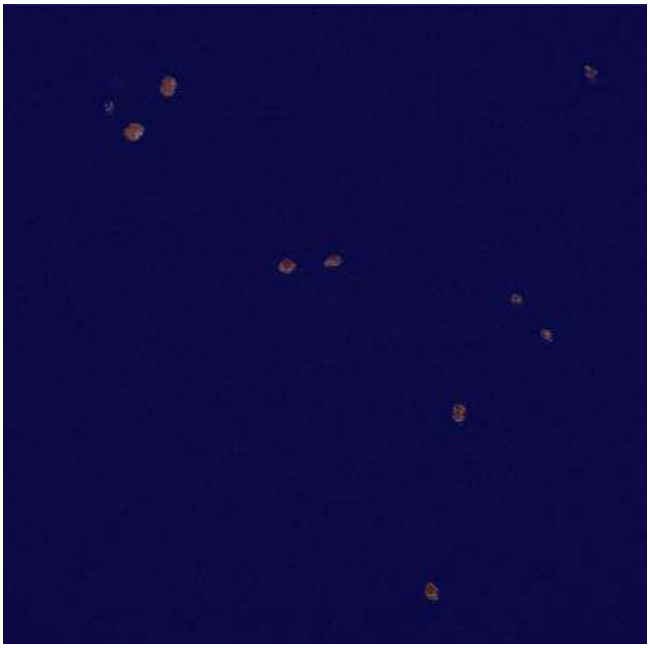
Segmentación de imágenes biomédicas con U-Net en el dataset Fluo-N2DL-HeLa



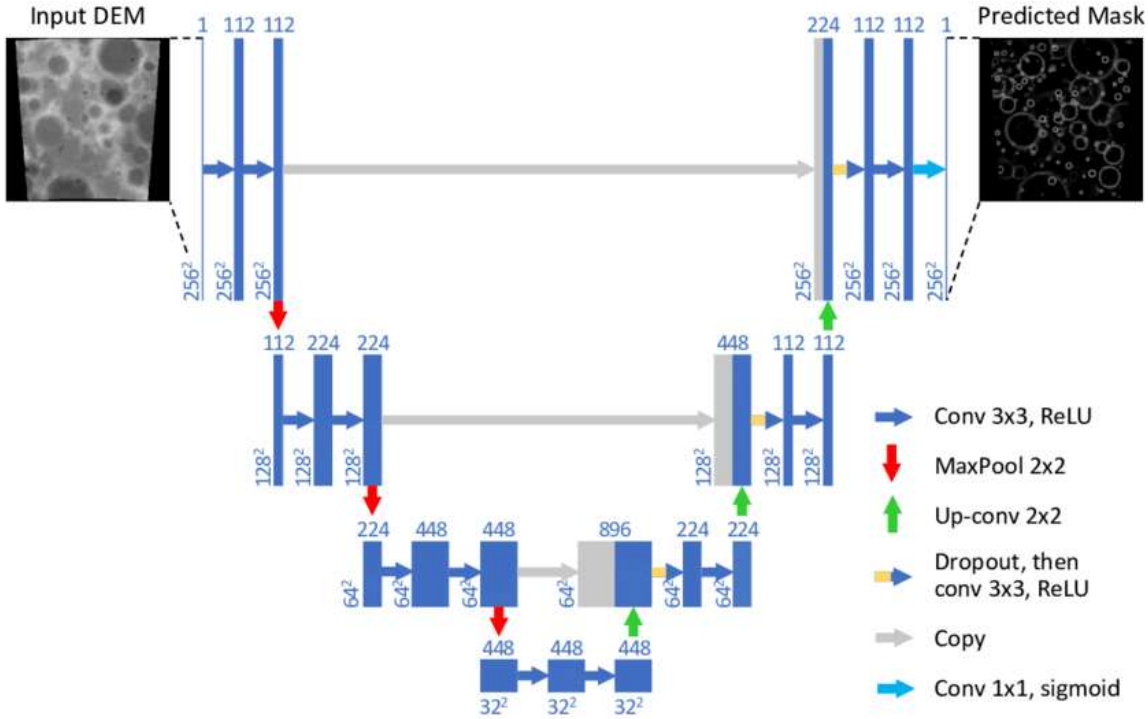
1. Introducción

La segmentación de imágenes biomédicas consiste en delimitar y clasificar cada píxel de una imagen de acuerdo a la estructura anatómica o celular que representa. En particular, la segmentación de núcleos celulares en imágenes de microscopía es un paso crucial para cuantificar proliferación y realizar seguimiento de células individuales en experimentos de time-lapse. Este proyecto aborda el problema de segmentar núcleos en imágenes fluorescentes de células HeLa (línea celular de cáncer cervical) expresando la proteína H2B-GFP (que marca los núcleos). Para ello, utilizaremos la arquitectura de red neuronal convolucional U-Net propuesta por Ronneberger et al. (2015), dado que está especialmente diseñada para segmentación biomédica y ha demostrado lograr resultados sobresalientes incluso con conjuntos de datos de tamaño moderado. En este cuaderno explicaremos detalladamente la teoría detrás de U-Net y su arquitectura original, implementaremos un modelo U-Net en PyTorch ajustado al problema de segmentación de núcleos en el conjunto de datos Fluo-N2DL-HeLa, y evaluaremos su desempeño con métricas cuantitativas y visualización de resultados. El contenido está estructurado en las siguientes secciones principales:

- **Teoría y arquitectura de U-Net:** explicación didáctica de la arquitectura U-Net basada en el paper original, describiendo su funcionamiento interno y ventajas clave, con ejemplos ilustrativos.
- **Preparación del conjunto de datos (Fluo-N2DL-HeLa):** cómo descargar y organizar las secuencias de imágenes, cargar los datos en PyTorch y aplicar preprocesamiento y aumentos de datos apropiados.
- **Implementación práctica de U-Net en PyTorch:** construcción paso a paso del modelo U-Net, configuración del proceso de entrenamiento (incluyendo posible uso de modelos pre-entrenados o técnicas de transferencia de aprendizaje), y ejecución de la inferencia sobre imágenes de prueba.
- **Evaluación de la segmentación:** cálculo de métricas cuantitativas (como IoU) y visualización de las máscaras producidas por el modelo en comparación con las anotaciones reales.

El objetivo es que este notebook sirva como una guía completa, desde la teoría hasta la práctica, para cualquier persona con conocimientos básicos de redes neuronales convolucionales y visión por ordenador, aplicando estos conceptos a un caso real de segmentación de imágenes de microscopía. A lo largo del cuaderno se justifican las decisiones técnicas tomadas, se incluyen referencias a recursos y trabajos previos relevantes, y se ilustran los resultados de forma clara.

2. Teoría del modelo U-Net: arquitectura y funcionamiento



U-Net es una arquitectura de red neuronal convolucional de tipo encoder-decoder con forma de "U" simétrica, diseñada específicamente para la segmentación de imágenes biomédicas. Esta arquitectura fue introducida por Olaf Ronneberger y cols. en 2015 con la motivación de lograr segmentaciones precisas incluso cuando la cantidad de datos de entrenamiento es limitada. A continuación, describimos la arquitectura y las ideas clave que la hacen especialmente efectiva.

Arquitectura original de U-Net propuesta por Ronneberger et al. (2015). Consta de un camino de contracción (encoder, izquierda) y un camino de expansión (decoder, derecha) conectados mediante skip connections. Cada bloque del encoder aplica dos convoluciones 3×3 con función de activación ReLU (flechas azules) seguidas de una operación de pooling 2×2 que reduce la resolución espacial (flechas rojas). En el decoder, cada bloque consta de una operación de upsampling o convolución transpuesta 2×2 que duplica la resolución (flechas verdes), concatenación ("copy and crop") con la característica correspondiente del encoder (flechas grises), y dos convoluciones 3×3 + ReLU. Las cifras en la parte superior de las cajas indican el número de features maps (profundidad de canal) y las dimensiones x-y se indican en la esquina inferior izquierda de cada bloque.

Como se observa en la imagen, el camino de contracción (parte izquierda, encoder) es una serie de bloques de dos convoluciones seguidas de max pooling, lo cual reduce progresivamente el tamaño espacial de la imagen a la vez que aumenta la profundidad de las características aprendidas. Este encoder extrae características de nivel cada vez más alto (más abstractas) a costa de perder resolución espacial. Por ejemplo, en la configuración original, la imagen de entrada (una tile de 572×572 px en escala de grises) primero se convierte en 64 mapas de características de 568×568 tras dos convoluciones, luego se reduce a 284×284 por pooling, se convolve a 128 canales, y así sucesivamente.

Por otro lado, el camino de expansión (parte derecha, decoder) reconstruye una representación segmentada de la imagen volviendo a aumentar la resolución paso a paso. Cada etapa del decoder realiza una operación de upsampling (a menudo implementada como una convolución transpuesta 2×2) que duplica la resolución horizontal y vertical, seguida de convoluciones que refinan los detalles. Un elemento fundamental son las conexiones de salto (skip connections): en cada nivel, la salida intermedia correspondiente del encoder se copia y concatena ("skip") con la entrada del decoder del mismo nivel de resolución. Estas conexiones permiten transferir directamente características de bajo nivel (alta resolución) desde el encoder al decoder, aportando información espacial fina que de otro modo se habría perdido durante el pooling. Gracias a ello, el decoder puede recuperar detalles precisos de bordes y contornos que son cruciales para segmentar correctamente estructuras pequeñas como núcleos celulares.

Las skip connections no solo ayudan a preservar la información de alta resolución, sino que también mejoran el flujo de gradientes durante el entrenamiento. Al crear rutas cortas entre las capas iniciales (encoder) y las finales (decoder), facilitan la propagación de gradientes hacia las primeras capas, mitigando el problema de gradiente desvaneciente en redes profundas.

U-Net combina efectivamente un contexto global (a través del encoder que capta la estructura general de la imagen) con una localización precisa (a través del decoder y los skips que recuperan los detalles locales), resultando en segmentaciones pixel a pixel muy exactas.

2.1 Ventajas de U-Net: Las características descritas hacen que U-Net tenga varias ventajas importantes en segmentación biomédica:

- **Alta precisión espacial:** Las skip connections permiten que el modelo aproveche tanto las características de alto nivel como la información de bordes finos, logrando delinear con exactitud incluso objetos pequeños o débiles en la imagen (por ejemplo, los contornos de núcleos celulares adyacentes).
- **Eficiente con pocos datos:** U-Net fue concebida para escenarios con datos limitados. Mediante un uso intensivo de data augmentation (transformaciones aleatorias de las imágenes de entrenamiento) y la arquitectura especializada, puede entrenarse end-to-end con relativamente pocas imágenes anotadas y aun así generalizar bien. En el trabajo original, por ejemplo, el modelo alcanzó resultados superiores al estado del arte entrenando con solo decenas de imágenes por clase.
- **Entrenamiento e inferencia rápidos:** A pesar de su profundidad, U-Net es eficiente. Los autores reportan que segmentar una imagen de ~512×512 px toma menos de 1 segundo en una GPU moderna, lo cual la hace viable para análisis de grandes pilas de imágenes. Además, su entrenamiento con imágenes de tamaño moderado es manejable en hardware estándar gracias al tamaño de lote pequeño y a la reutilización de características por las conexiones de salto.
- **Flexibilidad y extensión:** U-Net es una arquitectura plenamente convolucional, lo que significa que puede adaptarse a diferentes tamaños de entrada (mediante padding o subdividiendo la imagen en tiles) y a distintos números de clases de segmentación simplemente cambiando el número de mapas de salida en la última capa. Esto la ha vuelto una base popular para numerosas variantes y aplicaciones, incluyendo segmentación multi-clase, 3D U-Net para volumétricas, ResU-Net (incorporando residual blocks), U-Net++ con densos skip connections, entre otras.

U-Net ofrece un equilibrio ideal entre precisión y eficiencia para segmentación biomédica. Emplearemos esta arquitectura para nuestro problema de segmentar núcleos de células HeLa en imágenes de fluorescencia, aprovechando sus ventajas para lidiar con la escasez de datos anotados y la necesidad de segmentación detallada a nivel de píxel.

3. Preparación del dataset Fluo-N2DL-HeLa

Para entrenar y evaluar nuestro modelo, utilizamos el conjunto de datos Fluo-N2DL-HeLa. Este dataset consiste en secuencias de imágenes microscópicas fluorescentes 2D + tiempo (time-lapse) de núcleos celulares: en concreto, son películas de células HeLa (células de cáncer cervicouterino humano) que expresan de forma estable la histona H2B fusionada a GFP, lo que marca el ADN nuclear con fluorescencia verde. Cada secuencia es una serie temporal de imágenes tomadas bajo el microscopio con un intervalo de 30 minutos entre cuadros, cubriendo aproximadamente 46 horas de duración total. En total, el dataset proporciona 2 secuencias de entrenamiento con sus anotaciones de referencia y 2 secuencias de prueba (sin anotar). Cada secuencia de entrenamiento contiene 92 frames (imágenes) en escala de grises (un solo canal, correspondiente a la intensidad de fluorescencia GFP)

El archivo de entrenamiento (Fluo-N2DL-HeLa.zip) contiene dos tipos de carpetas nombradas 01 y 02 correspondientes a las dos secuencias de entrenamiento. Dentro de cada carpeta, las imágenes sin anotar están numeradas consecutivamente como t000.tif, t001.tif, ..., hasta t091.tif (92 frames). Adicionalmente, hay una subcarpeta 01_ST (y 02_ST respectivamente) que incluye las anotaciones de segmentación (SEG). Para nuestro objetivo de entrenamiento de U-Net usaremos las máscaras de segmentación (SEG), donde cada píxel de núcleo tiene un valor entero >0 (que identifica el objeto/célula) y el fondo es 0. Podemos convertir estas máscaras de instancia en máscaras binarias fácilmente (píxeles de núcleos con valor 1, fondo con 0) para entrenar una segmentación binaria (núcleo vs. fondo).

1) Se definen las rutas base para el dataset, usando las carpetas "01" y "01_ST/SEG" y "02" y "02_ST/SEG". Esto garantiza que cada imagen se empareje correctamente con su máscara correspondiente.

2) Cada imagen se carga usando PIL y se convierte a un array NumPy. Se transforma de uint16 a float32 y se normaliza, para que los valores queden en el rango [0,1].

3) Se convierten los arrays a tensores de PyTorch y se añade una dimensión de canal (resultado: (1, H, W)). Esto es fundamental para que el modelo trabaje correctamente con imágenes monocanales.

4) Se crean DataLoaders para iterar en batches durante el entrenamiento y la evaluación. Además, se incluye un ejemplo de visualización que muestra la imagen, la máscara y la superposición de ambas usando el colormap "viridis" para facilitar la comprobación visual del preprocesamiento.

In [ ]:
```
import os
from pathlib import Path
import shutil

base_dir = Path(r"C:\Users\Jm\Desktop\MASTER MII\PRIMERO\VISION POR ORDENADOR\Fluo-N2DL-HeLa(train)")
img_dirs = [base_dir / "01", base_dir / "02"]
mask_dirs = [base_dir / "01_ST" / "SEG", base_dir / "02_ST" / "SEG"]

# Creamos nueva estructura de directorios
new_dataset_dir = base_dir / "combined_dataset"
new_img_dir = new_dataset_dir / "images"
new_mask_dir = new_dataset_dir / "masks"

new_img_dir.mkdir(parents=True, exist_ok=True)
new_mask_dir.mkdir(parents=True, exist_ok=True)

# Función para copiar y renombrar archivos de una lista de directorios
def copiar_y_renombrar(dirs, extension, nuevo_directorio, prefijo):
    contador = 0
    for d in dirs:
        # Asegurarse de que la carpeta existe
        if not d.exists():
            print(f"La carpeta {d} no existe.")
            continue
        for archivo in sorted(os.listdir(d)):
            if archivo.lower().endswith(extension):
                ruta_origen = d / archivo
                nuevo_nombre = f"{prefijo}_{contador:03d}{extension}"
                ruta_destino = nuevo_directorio / nuevo_nombre
                shutil.copy(ruta_origen, ruta_destino)
                contador += 1
    print(f"Se copiaron {contador} archivos en {nuevo_directorio}")

# Copiamos imágenes (archivos .tif) y máscaras
copiar_y_renombrar(img_dirs, '.tif', new_img_dir, "img")
copiar_y_renombrar(mask_dirs, '.tif', new_mask_dir, "mask")

print("Se ha creado la nueva estructura de dataset en:", new_dataset_dir)
```

```
Se copiaron 184 archivos en C:\Users\Jm\Desktop\MASTER MII\PRIMERO\VISION POR ORDENADOR\Fluo-N2DL-HeLa(train)\combined_dataset\images
Se copiaron 184 archivos en C:\Users\Jm\Desktop\MASTER MII\PRIMERO\VISION POR ORDENADOR\Fluo-N2DL-HeLa(train)\combined_dataset\masks
Se ha creado la nueva estructura de dataset en: C:\Users\Jm\Desktop\MASTER MII\PRIMERO\VISION POR ORDENADOR\Fluo-N2DL-HeLa(train)\combined_dataset
```

In [64]:
```
import os
from pathlib import Path
import numpy as np
import torch
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
from PIL import Image

# Función de preprocesado (canal verde y normalización)
def preprocesar_frame(frame):
    if frame.ndim == 3:
        canal = frame[:, :, 1]
    else:
        canal = frame
    canal = canal.astype(np.float32)
    mn, mx = canal.min(), canal.max()
    if mx > mn:
```

```
        return (canal - mn) / (mx - mn)
    else:
        # evitamos la división por cero
        return canal / (canal.max() + 1e-6)

class FluoHeLa_TIFF_Dataset(Dataset):
    def __init__(self, img_dir, mask_dir, transform=None):

        self.img_paths  = sorted(Path(img_dir).glob("*.tif"))
        self.mask_paths = sorted(Path(mask_dir).glob("*.tif"))
        assert len(self.img_paths) == len(self.mask_paths), \
               "Número de imágenes y máscaras no coincide"
        self.transform = transform

    def __len__(self):
        return len(self.img_paths)

    def __getitem__(self, idx):
        # 1) Leemos frame multicanal o grayscale con PIL
        frame_pil = Image.open(self.img_paths[idx])
        frame_np  = np.array(frame_pil)
        img_np    = preprocesar_frame(frame_np)

        # 2) Leemos y binarizamos la máscara
        mask_pil  = Image.open(self.mask_paths[idx])
        mask_np   = np.array(mask_pil)
        mask_np   = (mask_np > 0).astype(np.float32)

        # 4) Convertimos a tensores con canal
        img_tensor  = torch.from_numpy(img_np).unsqueeze(0)
        mask_tensor = torch.from_numpy(mask_np).unsqueeze(0)

        return img_tensor, mask_tensor


if __name__ == "__main__":
    base_path       = Path("Fluo-N2DL-HeLa(train)")
    train_img_dir   = base_path / "combined_dataset"      / "images"
    train_mask_dir  = base_path / "combined_dataset"      / "masks"
    test_img_dir    = base_path / "combined_dataset_test" / "images"
    test_mask_dir   = base_path / "combined_dataset_test" / "masks"

    train_dataset = FluoHeLa_TIFF_Dataset(train_img_dir, train_mask_dir)
    test_dataset  = FluoHeLa_TIFF_Dataset(test_img_dir,  test_mask_dir)

    train_loader = DataLoader(train_dataset, batch_size=4, shuffle=True,  pin_memory=True)
    test_loader  = DataLoader(test_dataset,  batch_size=8, shuffle=False, pin_memory=True)

    print("Número de imágenes de entrenamiento:", len(train_dataset))
    print("Número de imágenes de test:", len(test_dataset))

    imgs, masks = next(iter(train_loader))
    print("Batch images shape:", imgs.shape)
    print("Batch masks shape:", masks.shape)
```

```
Número de imágenes de entrenamiento: 170
Número de imágenes de test: 14
Batch images shape: torch.Size([4, 1, 700, 1100])
Batch masks shape: torch.Size([4, 1, 700, 1100])
```

In [65]:
```
dataloader = { 'train': train_loader, 'test': test_loader}
```

El siguiente paso después de crear los datasets y dataloaders es ocnfirmar que todo se encuentra correctamente. La mejor forma que tenemos que nos permite confirmar que el preprocesamiento se ha realizado correctamente y que las imágenes y máscaras están en el formato esperado para el entrenamiento del modelo es mostrando las imágenes, para ello:

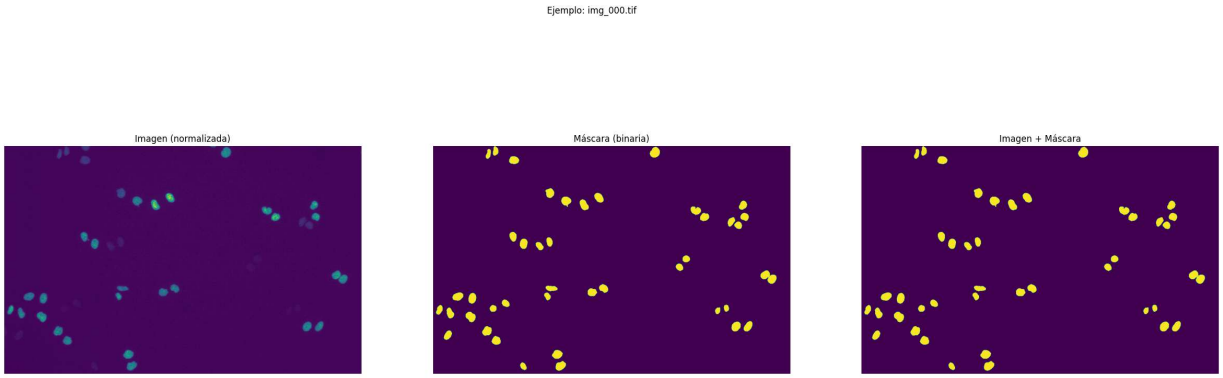
1) Se usan squeeze() para remover la dimensión de canal, quedando la imagen y la máscara en forma (H, W) para facilitar su visualización.

2) Visualizamos en 3 subplots:

- El primer subplot muestra la imagen normalizada en escala de grises.

- El segundo muestra la máscara binaria en escala de grises.
- El tercer subplot superpone la máscara (con colormap "viridis" y transparencia) sobre la imagen original, lo que facilita comprobar visualmente la correspondencia entre ambas.

In [66]:
```
# Visualizamos un ejemplo del conjunto de entrenamiento
img_tensor, mask_tensor = train_dataset[0]
img_np = img_tensor.squeeze().numpy()
mask_np = mask_tensor.squeeze().numpy()

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(30, 10))
ax1.imshow(img_np)
ax1.set_title("Imagen (normalizada)")
ax1.axis('off')

ax2.imshow(mask_np)
ax2.set_title("Máscara (binaria)")
ax2.axis('off')

ax3.imshow(img_np)
ax3.imshow(mask_np)
ax3.set_title("Imagen + Máscara")
ax3.axis('off')

plt.suptitle(f"Ejemplo: {train_dataset.img_paths[0].name}")
plt.show()
```



Nuestras imágenes tienen 700 x 1100 píxeles, almacenadas como arrays de NumPy (que podemos cargar con la función np.load). Ya están normalizadas y en formato float32.

In [4]:
```
img_np.shape, img_np.dtype, img_np.max(), img_np.min()
```

Out[4]:
```
((700, 1100), dtype('float32'), np.float32(1.0), np.float32(0.0))
```

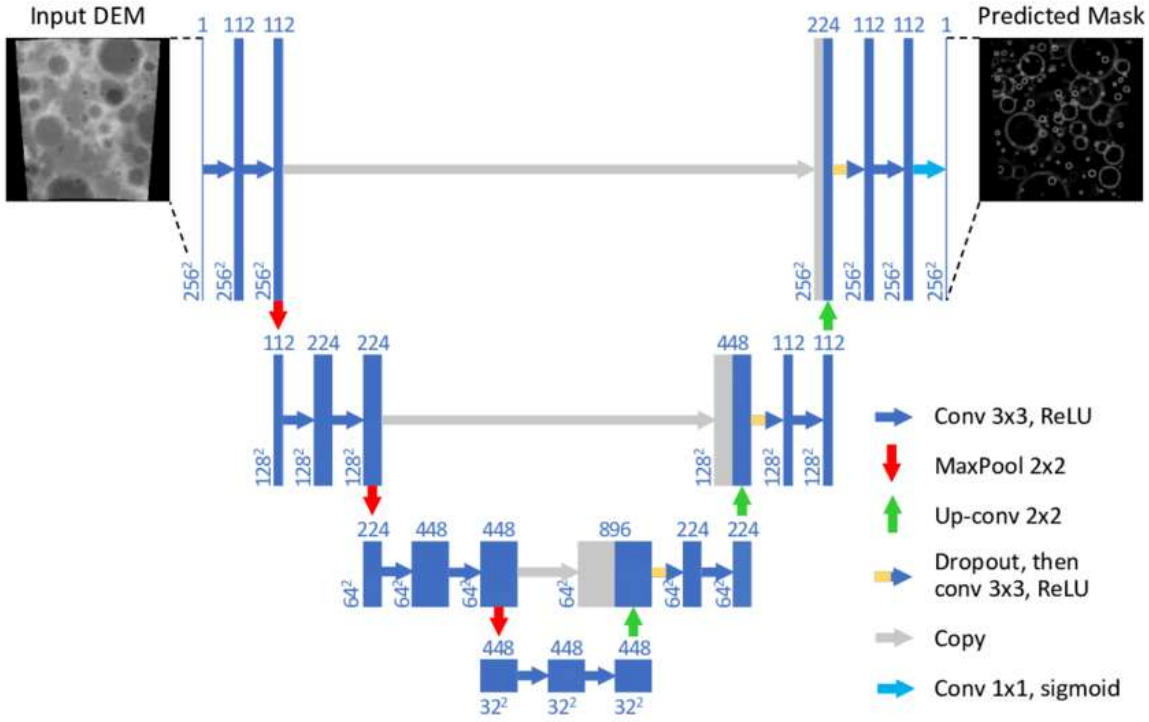
En cuanto a las máscaras, también las tenemos guardadas como arrays de NumPy. La resolución es la misma que las de la imagen original. En cada píxel podemos encontrar dos posibles valores: 0 o 1 . Este valor indica la clase (background o objeto).

In [5]:
```
mask_np.shape, mask_np.dtype, mask_np.max(), mask_np.min()
```

Out[5]:
```
((700, 1100), dtype('float32'), np.float32(1.0), np.float32(0.0))
```

Otro aspecto interesante de este estudio es que en este caso no necesitamos aplicar OneHot Encoding ya que solo tenemos dos clases y podemos solventar este problema aplicando binarización.

4. Implementación del modelo U-Net en PyTorch



Tras explicar anteriormente de forma teórica como funciona la arquitectura U-net, es hora de ver como la he implementado en PyTorch de forma sencilla sin usar ninguna implentación ya hecha. Esta implemntación, al igual que en la arquitectura, se divide por bloques donde cada uno tiene una función concreta. Estos bloques, por orden, son:

Bloque 1: Bloque de Convolución 3×3 con BatchNorm y ReLU (conv3x3_bn)

Esta función crea un bloque secuencial que aplica una convolución 3×3 (con padding=1 para conservar las dimensiones espaciales), seguida de una normalización por lotes y una activación ReLU. Se utiliza para extraer características manteniendo la resolución espacial. En la imagen, las dos flechas azules que unen los dos bloques también azules.

Bloque 2: Bloque de Encoder (encoder_conv)

Este bloque comienza con un max-pooling (reduciendo la resolución a la mitad) y luego aplica dos bloques conv3x3_bn. Se utiliza en la parte de encoder para extraer características de mayor nivel mientras se reduce la dimensión espacial de la imagen. En la imagen, simula el funcionamiento de un bloque entero, desde la flecha roja (max-pooling) hasta las dos flechas azules, la implementación anterior.

Bloque 3: Bloque de Decoder (clase deconv)

La clase deconv implementa un bloque de upsampling en el decoder. Primero, se realiza un upsample mediante una convolución transpuesta (ConvTranspose2d) que duplica la resolución. Luego, se ajusta la diferencia de tamaño entre el mapa de características upsampleado y la salida correspondiente del encoder usando F.pad. Finalmente, se concatenan ambos tensores a lo largo del canal y se aplican dos bloques conv3x3_bn para refinar las características y recuperar detalles espaciales. En la imagen, simula la flecha verde que apunta hacia arriba(up-conv) seguido de la misma implentación de antes usando las flechas azules.

Bloque 4: Arquitectura completa UNet

La clase UNet define la red completa. En el constructor, se definen:

- Una capa inicial que procesa la imagen de entrada con dos bloques conv3x3_bn.
- Varias capas de encoder (conv2, conv3, conv4) que son cada uno de los bloques de la imagen, que reducen progresivamente la resolución y aumentan la profundidad de las características.
- Capas de decoder (deconv1, deconv2, deconv3) que realizan upsampling y combinan las características del encoder a través de skip connections.

- Una última capa de convolución (self.out) que reduce el número de canales a n_classes (en segmentación binaria, n_classes=1).

El método forward pasa la imagen por el encoder, guarda las salidas intermedias para los saltos, y luego reconstruye la resolución mediante el decoder, obteniendo finalmente la máscara de segmentación.

Bloque 5: Diseño y Parámetros

El modelo utiliza una lista de filtros [16, 32, 64, 128] para cada nivel del encoder, lo que permite capturar tanto detalles finos como información contextual. La simetría entre encoder y decoder, junto con las skip connections, permite que la red combine información global y local, resultando en segmentaciones precisas.

Esta estructura sigue el diseño original de U-Net y se adapta para tareas de segmentación binaria, donde se espera que la salida tenga un solo canal si se define n_classes=1. Cada bloque está diseñado para optimizar la extracción y recuperación de información, asegurando que la red aprenda tanto características de alto nivel como detalles espaciales críticos para la segmentación de células.

In [67]:
```
import torch
import torch.nn.functional as F

def conv3x3_bn(ci, co):
    return torch.nn.Sequential(
        torch.nn.Conv2d(ci, co, 3, padding=1),
        torch.nn.BatchNorm2d(co),
        torch.nn.ReLU(inplace=True)
    )

def encoder_conv(ci, co):
  return torch.nn.Sequential(
        torch.nn.MaxPool2d(2),
        conv3x3_bn(ci, co),
        conv3x3_bn(co, co),
    )

class deconv(torch.nn.Module):
    def __init__(self, ci, co):
        super(deconv, self).__init__()
        self.upsample = torch.nn.ConvTranspose2d(ci, co, 2, stride=2)
        self.conv1 = conv3x3_bn(ci, co)
        self.conv2 = conv3x3_bn(co, co)

    # Recibe la salida de la capa anterior y la salida de la etapa
    def forward(self, x1, x2):
        x1 = self.upsample(x1)
        diffX = x2.size()[2] - x1.size()[2]
        diffY = x2.size()[3] - x1.size()[3]
        x1 = F.pad(x1, (diffX, 0, diffY, 0))
        # concatenamos los tensores
        x = torch.cat([x2, x1], dim=1)
        x = self.conv1(x)
        x = self.conv2(x)
        return x

class UNet(torch.nn.Module):
    def __init__(self, n_classes=1, in_ch=1):
        super().__init__()

        # Lista de capas en encoder-decoder con número de filtros
        c = [16, 32, 64, 128]

        # Primera capa conv que recibe la imagen
        self.conv1 = torch.nn.Sequential(
          conv3x3_bn(in_ch, c[0]),
          conv3x3_bn(c[0], c[0]),
        )
        # Capas del encoder
        self.conv2 = encoder_conv(c[0], c[1])
        self.conv3 = encoder_conv(c[1], c[2])
        self.conv4 = encoder_conv(c[2], c[3])

        # Capas del decoder
        self.deconv1 = deconv(c[3],c[2])
```

```
        self.deconv2 = deconv(c[2],c[1])
        self.deconv3 = deconv(c[1],c[0])

        # Útlima capa conv que nos da la máscara
        self.out = torch.nn.Conv2d(c[0], n_classes, 3, padding=1)

    def forward(self, x):
        # Encoder
        x1 = self.conv1(x)
        x2 = self.conv2(x1)
        x3 = self.conv3(x2)
        x = self.conv4(x3)
        # Decoder
        x = self.deconv1(x, x3)
        x = self.deconv2(x, x2)
        x = self.deconv3(x, x1)
        x = self.out(x)
        return x
```

Comprobamos que estamos haciendo todo bien pasándole a la arquitectura un tensor de prueba con las medidas que debería de tener. Vemos que el resultado es correcto, un tensor con un batch size de 10, un único canal ya que es en blanco y negro y la resolución de nuestras imagenes 700x1100.

In [7]:
```
model = UNet()
output = model(torch.randn((10,1,700,1100)))
output.shape
```

Out[7]:
```
torch.Size([10, 1, 700, 1100])
```

4.1 Fit de 1 única muestra

Para comprobar que todo funciona hacemos el fit de una sola muestra. Usamos la función de pérdida BCEWithLogitsLoss que viene bien en estos casos para optimizar la red, que aplicará la función de activación sigmoid a las salidas de la red (binaria para que estén entre 0 y 1) y luego, finalmente, calcula la función binary cross entropy.

In [8]:
```
device = "cuda" if torch.cuda.is_available() else "cpu"
```

In [9]:
```
def fit(model, X, y, epochs=1, lr=3e-4):
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = torch.nn.BCEWithLogitsLoss()
    model.to(device)
    X, y = X.to(device), y.to(device)
    model.train()
    for epoch in range(1, epochs+1):
        optimizer.zero_grad()
        y_hat = model(X)
        loss = criterion(y_hat, y)
        loss.backward()
        optimizer.step()
        print(f"Epoch {epoch}/{epochs} loss {loss.item():.5f}")
```

In [68]:
```
img_tensor = torch.tensor(img_np).unsqueeze(0).unsqueeze(0)
mask_tensor = torch.tensor(mask_np, dtype=torch.float32).unsqueeze(0).unsqueeze(0)

img_tensor.shape, mask_tensor.shape
```

Out[68]:
```
(torch.Size([1, 1, 700, 1100]), torch.Size([1, 1, 700, 1100]))
```

In [11]:
```
fit(model, img_tensor, mask_tensor, epochs=20)
```

```
Epoch 1/20 loss 0.69655
Epoch 2/20 loss 0.67303
Epoch 3/20 loss 0.65620
Epoch 4/20 loss 0.64202
Epoch 5/20 loss 0.62767
Epoch 6/20 loss 0.61232
Epoch 7/20 loss 0.59791
Epoch 8/20 loss 0.58442
Epoch 9/20 loss 0.57106
Epoch 10/20 loss 0.55855
Epoch 11/20 loss 0.54742
Epoch 12/20 loss 0.53742
Epoch 13/20 loss 0.52807
Epoch 14/20 loss 0.51914
Epoch 15/20 loss 0.51054
Epoch 16/20 loss 0.50218
Epoch 17/20 loss 0.49399
Epoch 18/20 loss 0.48599
Epoch 19/20 loss 0.47820
Epoch 20/20 loss 0.47060
```

La función de pérdida baja progresivamente, así que parece que está funcionando bien. No obstante, necesitamos alguna métrica para evaluar cuánto se parecen las máscaras predichas a las reales. Para ello usamos la métrica IoU, Intersection Over Union, y que calcula la relación entre la intersección y la unión de dos áreas.

In [12]:
```python
def iou_binary(outputs, labels, smooth=1e-6):
    # Convertir las salidas a binario
    outputs = (torch.sigmoid(outputs) > 0.5).float()
    labels = (labels > 0.5).float()
    intersection = (outputs * labels).sum(dim=(1,2,3))
    union = outputs.sum(dim=(1,2,3)) + labels.sum(dim=(1,2,3)) - intersection
    iou = (intersection + smooth) / (union + smooth)
    return iou.mean().item()
```

Volviendo a entrenar de nuevo con la métrica de IoU

In [13]:
```python
def fit(model, X, y, epochs=1, lr=1e-3):
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = torch.nn.BCEWithLogitsLoss()
    model.to(device)
    X, y = X.to(device), y.to(device)
    model.train()
    for epoch in range(1, epochs+1):
        optimizer.zero_grad()
        y_hat = model(X)
        loss = criterion(y_hat, y)
        loss.backward()
        optimizer.step()
        current_iou = iou_binary(y_hat, y)
        print(f"Epoch {epoch}/{epochs} loss {loss.item():.5f} iou {current_iou:.5f}")
```

El resultado debería de ser casi perfecto o perfecto 1.00 si lo dejamos el suficiente tiempo, pero realmente solo queremos saber si funciona bien la arquitectura creada, por lo que con un resultado correcto nos conformaremos.

In [14]:
```python
fit(model, img_tensor, mask_tensor, epochs=100)
```

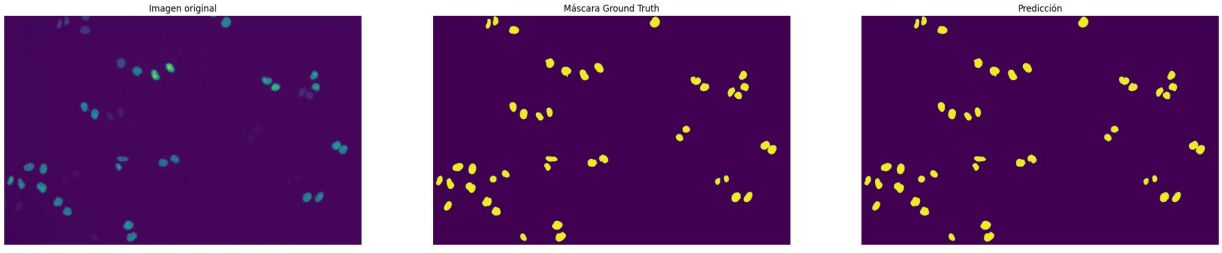
```
Epoch 1/100 loss 0.46316 iou 0.73036
Epoch 2/100 loss 0.44519 iou 0.73917
Epoch 3/100 loss 0.41573 iou 0.74128
Epoch 4/100 loss 0.39627 iou 0.73656
Epoch 5/100 loss 0.37722 iou 0.74383
Epoch 6/100 loss 0.35783 iou 0.74727
Epoch 7/100 loss 0.33890 iou 0.75090
Epoch 8/100 loss 0.32422 iou 0.74817
Epoch 9/100 loss 0.30836 iou 0.74485
Epoch 10/100 loss 0.29288 iou 0.74013
Epoch 11/100 loss 0.27877 iou 0.74042
Epoch 12/100 loss 0.26446 iou 0.74389
Epoch 13/100 loss 0.25043 iou 0.74819
Epoch 14/100 loss 0.23778 iou 0.75447
Epoch 15/100 loss 0.22565 iou 0.76161
Epoch 16/100 loss 0.21384 iou 0.76657
Epoch 17/100 loss 0.20338 iou 0.76748
Epoch 18/100 loss 0.19394 iou 0.76805
Epoch 19/100 loss 0.18478 iou 0.77292
Epoch 20/100 loss 0.17608 iou 0.77986
Epoch 21/100 loss 0.16796 iou 0.78771
Epoch 22/100 loss 0.16002 iou 0.79177
Epoch 23/100 loss 0.15226 iou 0.79143
Epoch 24/100 loss 0.14494 iou 0.79243
Epoch 25/100 loss 0.13787 iou 0.79753
Epoch 26/100 loss 0.13094 iou 0.80465
Epoch 27/100 loss 0.12431 iou 0.81103
Epoch 28/100 loss 0.11795 iou 0.81559
Epoch 29/100 loss 0.11183 iou 0.82130
Epoch 30/100 loss 0.10604 iou 0.82984
Epoch 31/100 loss 0.10068 iou 0.84017
Epoch 32/100 loss 0.09574 iou 0.84743
Epoch 33/100 loss 0.09119 iou 0.85671
Epoch 34/100 loss 0.08706 iou 0.86454
Epoch 35/100 loss 0.08327 iou 0.87189
Epoch 36/100 loss 0.07973 iou 0.87981
Epoch 37/100 loss 0.07645 iou 0.88676
Epoch 38/100 loss 0.07339 iou 0.89362
Epoch 39/100 loss 0.07047 iou 0.89921
Epoch 40/100 loss 0.06769 iou 0.90486
Epoch 41/100 loss 0.06505 iou 0.90895
Epoch 42/100 loss 0.06253 iou 0.91194
Epoch 43/100 loss 0.06015 iou 0.91590
Epoch 44/100 loss 0.05793 iou 0.92197
Epoch 45/100 loss 0.05582 iou 0.92669
Epoch 46/100 loss 0.05379 iou 0.92933
Epoch 47/100 loss 0.05188 iou 0.92986
Epoch 48/100 loss 0.05009 iou 0.93157
Epoch 49/100 loss 0.04840 iou 0.93461
Epoch 50/100 loss 0.04680 iou 0.93744
Epoch 51/100 loss 0.04528 iou 0.93817
Epoch 52/100 loss 0.04382 iou 0.93962
Epoch 53/100 loss 0.04245 iou 0.94013
Epoch 54/100 loss 0.04111 iou 0.94163
Epoch 55/100 loss 0.03981 iou 0.94431
Epoch 56/100 loss 0.03858 iou 0.94533
Epoch 57/100 loss 0.03746 iou 0.94894
Epoch 58/100 loss 0.03631 iou 0.94822
Epoch 59/100 loss 0.03525 iou 0.94900
Epoch 60/100 loss 0.03412 iou 0.95361
Epoch 61/100 loss 0.03328 iou 0.95003
Epoch 62/100 loss 0.03242 iou 0.94674
Epoch 63/100 loss 0.03152 iou 0.94714
Epoch 64/100 loss 0.03043 iou 0.95789
Epoch 65/100 loss 0.02966 iou 0.95686
Epoch 66/100 loss 0.02880 iou 0.95906
Epoch 67/100 loss 0.02800 iou 0.96153
Epoch 68/100 loss 0.02732 iou 0.96209
Epoch 69/100 loss 0.02669 iou 0.96101
Epoch 70/100 loss 0.02600 iou 0.96598
Epoch 71/100 loss 0.02534 iou 0.96553
Epoch 72/100 loss 0.02477 iou 0.96550
Epoch 73/100 loss 0.02414 iou 0.96885
Epoch 74/100 loss 0.02361 iou 0.96944
Epoch 75/100 loss 0.02311 iou 0.97003
```

```
Epoch 76/100 loss 0.02264 iou 0.96881
Epoch 77/100 loss 0.02222 iou 0.96786
Epoch 78/100 loss 0.02186 iou 0.96519
Epoch 79/100 loss 0.02126 iou 0.96988
Epoch 80/100 loss 0.02076 iou 0.97351
Epoch 81/100 loss 0.02041 iou 0.97243
Epoch 82/100 loss 0.02012 iou 0.97003
Epoch 83/100 loss 0.01975 iou 0.96854
Epoch 84/100 loss 0.01927 iou 0.97307
Epoch 85/100 loss 0.01888 iou 0.97596
Epoch 86/100 loss 0.01862 iou 0.97257
Epoch 87/100 loss 0.01827 iou 0.97501
Epoch 88/100 loss 0.01790 iou 0.97628
Epoch 89/100 loss 0.01762 iou 0.97484
Epoch 90/100 loss 0.01733 iou 0.97660
Epoch 91/100 loss 0.01703 iou 0.97705
Epoch 92/100 loss 0.01672 iou 0.97862
Epoch 93/100 loss 0.01644 iou 0.97895
Epoch 94/100 loss 0.01623 iou 0.97643
Epoch 95/100 loss 0.01594 iou 0.97826
Epoch 96/100 loss 0.01568 iou 0.97920
Epoch 97/100 loss 0.01546 iou 0.97926
Epoch 98/100 loss 0.01520 iou 0.98021
Epoch 99/100 loss 0.01496 iou 0.98025
Epoch 100/100 loss 0.01473 iou 0.98090
```

Ahora podemos generar predicciones para obtener máscaras de segmentación.

In [15]:
```python
model.eval()

print("Tensor de entrada:", img_tensor.shape, img_tensor.dtype)
with torch.no_grad():
    output = model(img_tensor.to(device))  # output shape: (1, 1, H, W)
    prob = torch.sigmoid(output)           # convertir logits a probabilidades
    pred_mask = (prob > 0.50).float()         # binarizamos la salida


# Visualizar
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(30,10))
ax1.imshow(img_np)
ax1.set_title("Imagen original")
ax1.axis('off')

ax2.imshow(mask_np)
ax2.set_title("Máscara Ground Truth")
ax2.axis('off')

ax3.imshow(pred_mask.squeeze().cpu().numpy())
ax3.set_title("Predicción")
ax3.axis('off')

plt.show()
```

```
Tensor de entrada: torch.Size([1, 1, 700, 1100]) torch.float32
```



4.2 Entrenando con todo el dataset

Una vez hemos validado que nuestra red es capaz de hacer el fit de una imagen, podemos entrenar la red con todo el dataset.

In [16]:
```python
from tqdm import tqdm

def fit(model, dataloader, epochs=100, lr=3e-5):
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
```

```
    criterion = torch.nn.BCEWithLogitsLoss()
    model.to(device)
    hist = {'loss': [], 'iou': [], 'test_loss': [], 'test_iou': []}
    for epoch in range(1, epochs+1):
      bar = tqdm(dataloader['train'])
      train_loss, train_iou = [], []
      model.train()
      for imgs, masks in bar:
        imgs, masks = imgs.to(device), masks.to(device)
        optimizer.zero_grad()
        y_hat = model(imgs)
        loss = criterion(y_hat, masks)
        loss.backward()
        optimizer.step()
        ious = iou_binary(y_hat, masks)
        train_loss.append(loss.item())
        train_iou.append(ious)
        bar.set_description(f"loss {np.mean(train_loss):.5f} iou {np.mean(train_iou):.5f}")
      hist['loss'].append(np.mean(train_loss))
      hist['iou'].append(np.mean(train_iou))
      bar = tqdm(dataloader['test'])
      test_loss, test_iou = [], []
      model.eval()
      with torch.no_grad():
        for imgs, masks in bar:
          imgs, masks = imgs.to(device), masks.to(device)
          y_hat = model(imgs)
          loss = criterion(y_hat, masks)
          ious = iou_binary(y_hat, masks)
          test_loss.append(loss.item())
          test_iou.append(ious)
          bar.set_description(f"test_loss {np.mean(test_loss):.5f} test_iou {np.mean(test_iou):.5f}")
      hist['test_loss'].append(np.mean(test_loss))
      hist['test_iou'].append(np.mean(test_iou))
      print(f"\nEpoch {epoch}/{epochs} loss {np.mean(train_loss):.5f} iou {np.mean(train_iou):.5f} test_
    return hist
```

In [17]:
```
model = UNet()
hist = fit(model, dataloader, epochs=30)
```

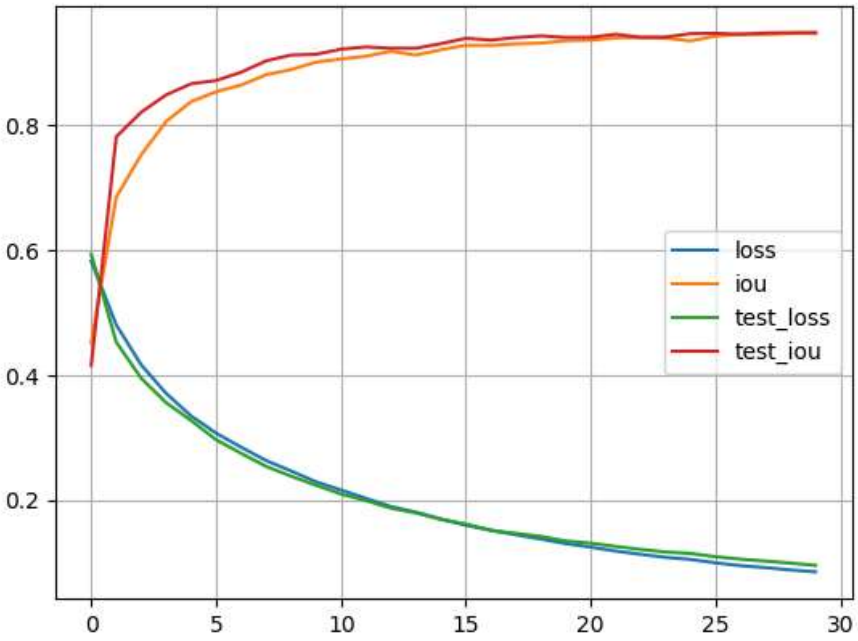
```
loss 0.58211 iou 0.45129: 100%|██████████| 43/43 [00:11<00:00,  3.82it/s]
test_loss 0.59283 test_iou 0.41513: 100%|██████████| 2/2 [00:00<00:00,  3.76it/s]
Epoch 1/30 loss 0.58211 iou 0.45129 test_loss 0.59283 test_iou 0.41513
loss 0.47992 iou 0.68425: 100%|██████████| 43/43 [00:08<00:00,  5.07it/s]
test_loss 0.45230 test_iou 0.78067: 100%|██████████| 2/2 [00:00<00:00,  5.61it/s]
Epoch 2/30 loss 0.47992 iou 0.68425 test_loss 0.45230 test_iou 0.78067
loss 0.41586 iou 0.75210: 100%|██████████| 43/43 [00:08<00:00,  4.91it/s]
test_loss 0.39440 test_iou 0.82016: 100%|██████████| 2/2 [00:00<00:00,  5.41it/s]
Epoch 3/30 loss 0.41586 iou 0.75210 test_loss 0.39440 test_iou 0.82016
loss 0.37011 iou 0.80529: 100%|██████████| 43/43 [00:08<00:00,  4.92it/s]
test_loss 0.35521 test_iou 0.84790: 100%|██████████| 2/2 [00:00<00:00,  5.39it/s]
Epoch 4/30 loss 0.37011 iou 0.80529 test_loss 0.35521 test_iou 0.84790
loss 0.33391 iou 0.83710: 100%|██████████| 43/43 [00:08<00:00,  4.93it/s]
test_loss 0.32679 test_iou 0.86567: 100%|██████████| 2/2 [00:00<00:00,  5.35it/s]
Epoch 5/30 loss 0.33391 iou 0.83710 test_loss 0.32679 test_iou 0.86567
loss 0.30656 iou 0.85287: 100%|██████████| 43/43 [00:08<00:00,  4.92it/s]
test_loss 0.29604 test_iou 0.87086: 100%|██████████| 2/2 [00:00<00:00,  5.40it/s]
Epoch 6/30 loss 0.30656 iou 0.85287 test_loss 0.29604 test_iou 0.87086
loss 0.28414 iou 0.86354: 100%|██████████| 43/43 [00:08<00:00,  4.90it/s]
test_loss 0.27430 test_iou 0.88414: 100%|██████████| 2/2 [00:00<00:00,  5.37it/s]
Epoch 7/30 loss 0.28414 iou 0.86354 test_loss 0.27430 test_iou 0.88414
loss 0.26270 iou 0.88005: 100%|██████████| 43/43 [00:08<00:00,  4.80it/s]
test_loss 0.25341 test_iou 0.90224: 100%|██████████| 2/2 [00:00<00:00,  5.29it/s]
Epoch 8/30 loss 0.26270 iou 0.88005 test_loss 0.25341 test_iou 0.90224
loss 0.24608 iou 0.88829: 100%|██████████| 43/43 [00:08<00:00,  4.85it/s]
test_loss 0.23770 test_iou 0.91155: 100%|██████████| 2/2 [00:00<00:00,  5.35it/s]
Epoch 9/30 loss 0.24608 iou 0.88829 test_loss 0.23770 test_iou 0.91155
loss 0.22867 iou 0.90037: 100%|██████████| 43/43 [00:08<00:00,  4.85it/s]
test_loss 0.22325 test_iou 0.91276: 100%|██████████| 2/2 [00:00<00:00,  5.24it/s]
Epoch 10/30 loss 0.22867 iou 0.90037 test_loss 0.22325 test_iou 0.91276
loss 0.21549 iou 0.90521: 100%|██████████| 43/43 [00:08<00:00,  4.83it/s]
test_loss 0.20912 test_iou 0.92099: 100%|██████████| 2/2 [00:00<00:00,  5.24it/s]
```

```
Epoch 11/30 loss 0.21549 iou 0.90521 test_loss 0.20912 test_iou 0.92099
loss 0.20220 iou 0.90968: 100%|██████████| 43/43 [00:08<00:00,  4.80it/s]
test_loss 0.19869 test_iou 0.92468: 100%|██████████| 2/2 [00:00<00:00,  5.23it/s]
Epoch 12/30 loss 0.20220 iou 0.90968 test_loss 0.19869 test_iou 0.92468
loss 0.18897 iou 0.91763: 100%|██████████| 43/43 [00:09<00:00,  4.77it/s]
test_loss 0.18652 test_iou 0.92242: 100%|██████████| 2/2 [00:00<00:00,  5.24it/s]
Epoch 13/30 loss 0.18897 iou 0.91763 test_loss 0.18652 test_iou 0.92242
loss 0.18014 iou 0.91152: 100%|██████████| 43/43 [00:09<00:00,  4.77it/s]
test_loss 0.17891 test_iou 0.92255: 100%|██████████| 2/2 [00:00<00:00,  5.05it/s]
Epoch 14/30 loss 0.18014 iou 0.91152 test_loss 0.17891 test_iou 0.92255
loss 0.16911 iou 0.92009: 100%|██████████| 43/43 [00:09<00:00,  4.76it/s]
test_loss 0.16862 test_iou 0.92977: 100%|██████████| 2/2 [00:00<00:00,  5.21it/s]
Epoch 15/30 loss 0.16911 iou 0.92009 test_loss 0.16862 test_iou 0.92977
loss 0.15908 iou 0.92713: 100%|██████████| 43/43 [00:09<00:00,  4.75it/s]
test_loss 0.16112 test_iou 0.93844: 100%|██████████| 2/2 [00:00<00:00,  5.17it/s]
Epoch 16/30 loss 0.15908 iou 0.92713 test_loss 0.16112 test_iou 0.93844
loss 0.15120 iou 0.92701: 100%|██████████| 43/43 [00:09<00:00,  4.75it/s]
test_loss 0.15100 test_iou 0.93555: 100%|██████████| 2/2 [00:00<00:00,  5.11it/s]
Epoch 17/30 loss 0.15120 iou 0.92701 test_loss 0.15100 test_iou 0.93555
loss 0.14395 iou 0.92951: 100%|██████████| 43/43 [00:09<00:00,  4.74it/s]
test_loss 0.14610 test_iou 0.93959: 100%|██████████| 2/2 [00:00<00:00,  5.10it/s]
Epoch 18/30 loss 0.14395 iou 0.92951 test_loss 0.14610 test_iou 0.93959
loss 0.13718 iou 0.93065: 100%|██████████| 43/43 [00:09<00:00,  4.74it/s]
test_loss 0.14126 test_iou 0.94230: 100%|██████████| 2/2 [00:00<00:00,  5.17it/s]
Epoch 19/30 loss 0.13718 iou 0.93065 test_loss 0.14126 test_iou 0.94230
loss 0.12974 iou 0.93450: 100%|██████████| 43/43 [00:09<00:00,  4.74it/s]
test_loss 0.13400 test_iou 0.93969: 100%|██████████| 2/2 [00:00<00:00,  5.19it/s]
Epoch 20/30 loss 0.12974 iou 0.93450 test_loss 0.13400 test_iou 0.93969
loss 0.12418 iou 0.93537: 100%|██████████| 43/43 [00:08<00:00,  4.79it/s]
test_loss 0.13047 test_iou 0.93991: 100%|██████████| 2/2 [00:00<00:00,  5.15it/s]
Epoch 21/30 loss 0.12418 iou 0.93537 test_loss 0.13047 test_iou 0.93991
loss 0.11786 iou 0.93906: 100%|██████████| 43/43 [00:08<00:00,  4.82it/s]
test_loss 0.12522 test_iou 0.94482: 100%|██████████| 2/2 [00:00<00:00,  5.14it/s]
Epoch 22/30 loss 0.11786 iou 0.93906 test_loss 0.12522 test_iou 0.94482
loss 0.11266 iou 0.93967: 100%|██████████| 43/43 [00:08<00:00,  4.79it/s]
test_loss 0.12049 test_iou 0.94000: 100%|██████████| 2/2 [00:00<00:00,  5.17it/s]
Epoch 23/30 loss 0.11266 iou 0.93967 test_loss 0.12049 test_iou 0.94000
loss 0.10776 iou 0.93866: 100%|██████████| 43/43 [00:08<00:00,  4.81it/s]
test_loss 0.11640 test_iou 0.94036: 100%|██████████| 2/2 [00:00<00:00,  5.11it/s]
Epoch 24/30 loss 0.10776 iou 0.93866 test_loss 0.11640 test_iou 0.94036
loss 0.10442 iou 0.93421: 100%|██████████| 43/43 [00:08<00:00,  4.81it/s]
test_loss 0.11402 test_iou 0.94591: 100%|██████████| 2/2 [00:00<00:00,  5.21it/s]
Epoch 25/30 loss 0.10442 iou 0.93421 test_loss 0.11402 test_iou 0.94591
loss 0.09901 iou 0.94205: 100%|██████████| 43/43 [00:08<00:00,  4.81it/s]
test_loss 0.10876 test_iou 0.94630: 100%|██████████| 2/2 [00:00<00:00,  5.19it/s]
Epoch 26/30 loss 0.09901 iou 0.94205 test_loss 0.10876 test_iou 0.94630
loss 0.09449 iou 0.94498: 100%|██████████| 43/43 [00:08<00:00,  4.81it/s]
test_loss 0.10487 test_iou 0.94490: 100%|██████████| 2/2 [00:00<00:00,  5.20it/s]
Epoch 27/30 loss 0.09449 iou 0.94498 test_loss 0.10487 test_iou 0.94490
loss 0.09134 iou 0.94455: 100%|██████████| 43/43 [00:09<00:00,  4.75it/s]
test_loss 0.10184 test_iou 0.94687: 100%|██████████| 2/2 [00:00<00:00,  5.13it/s]
Epoch 28/30 loss 0.09134 iou 0.94455 test_loss 0.10184 test_iou 0.94687
loss 0.08767 iou 0.94614: 100%|██████████| 43/43 [00:09<00:00,  4.75it/s]
test_loss 0.09857 test_iou 0.94734: 100%|██████████| 2/2 [00:00<00:00,  5.16it/s]
Epoch 29/30 loss 0.08767 iou 0.94614 test_loss 0.09857 test_iou 0.94734
loss 0.08457 iou 0.94620: 100%|██████████| 43/43 [00:09<00:00,  4.74it/s]
test_loss 0.09499 test_iou 0.94768: 100%|██████████| 2/2 [00:00<00:00,  5.18it/s]
Epoch 30/30 loss 0.08457 iou 0.94620 test_loss 0.09499 test_iou 0.94768
```

In [18]:
```python
# Guardamos el modelo
torch.save(model.state_dict(), "unet_model_final2.pth")
```

4.3 Análisis del entrenamiento con U-net

In [19]:
```python
import pandas as pd
df = pd.DataFrame(hist)
df.plot(grid=True)
plt.show()
```



Tras terminar el entrenamiento, los resultados muestran que la pérdida de entrenamiento disminuye de forma consistente y el IoU de entrenamiento mejora, lo que indica que la red está aprendiendo a segmentar bien las imágenes. Sin embargo, se observan algunas fluctuaciones en las métricas del conjunto de test, lo que sugiere que la generalización podría ser más estable. Esto es común en implementaciones personalizadas, donde el preprocesamiento y la cantidad de datos juegan un papel crucial.

La implementación que he realizado permite combinar información local y global para lograr segmentaciones precisas. La arquitectura es relativamente ligera, usando filtros que van de 16 a 128, lo que puede ser ventajoso en términos de tiempo de entrenamiento, pero a veces podría limitar la capacidad de capturar detalles muy finos o la variabilidad en datos complejos.

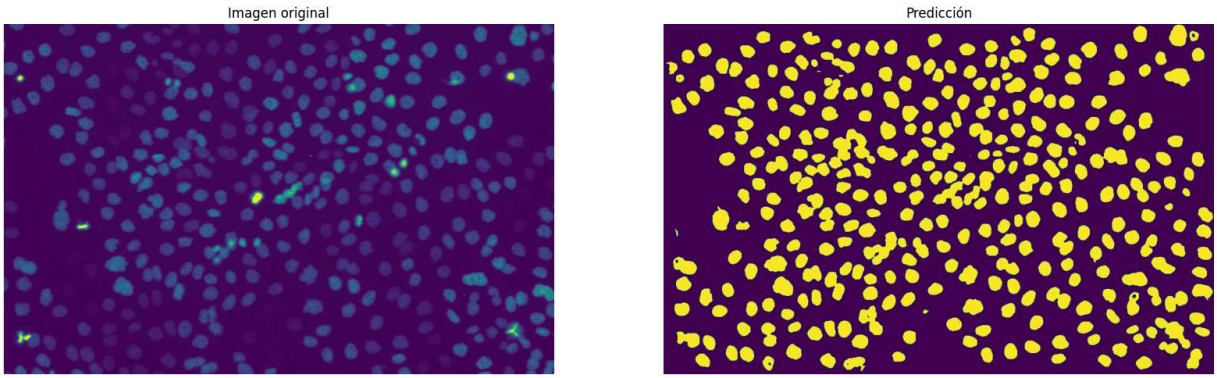
Como conclusión, podemos decir que la red U-Net que he creado es un excelente punto de partida para la segmentación de imágenes de células. He aprendido a segmentar correctamente, pero la variabilidad en el test sugiere que se puede afinar la arquitectura y el proceso de entrenamiento para lograr una generalización más robusta. Estas mejoras podrían incluir un aumento en la complejidad del modelo, una mayor regularización, o incluso incorporar técnicas y arquitecturas ya optimizadas de la comunidad.

4.4 Demostración del modelo creado por U-net en acción

Podemos ahora tomar todas las imágenes de prueba que tenemos y crear un video para ver si tienen sentido los resultados del entrenamiento. Para ello solo debemos aplicar la segmentación a cada uno de los frames que tenemos y unirlos.

In [76]:
```
# Visualizamos una imagen de test y su máscara predicha
model.load_state_dict(torch.load("unet_model_final2.pth", map_location=torch.device("cpu")))


img_tensor, _ = test_dataset[0]
img_tensor = img_tensor.unsqueeze(0).to(device)
output = model(img_tensor)
prob = torch.sigmoid(output)
pred_mask = (prob > 0.40).float()

# Visualizar
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))
ax1.imshow(img_tensor.squeeze().cpu().numpy())
ax1.set_title("Imagen original")

ax1.axis('off')
ax2.imshow(pred_mask.squeeze().cpu().numpy())
ax2.set_title("Predicción")
ax2.axis('off')
```

```
plt.show()
```

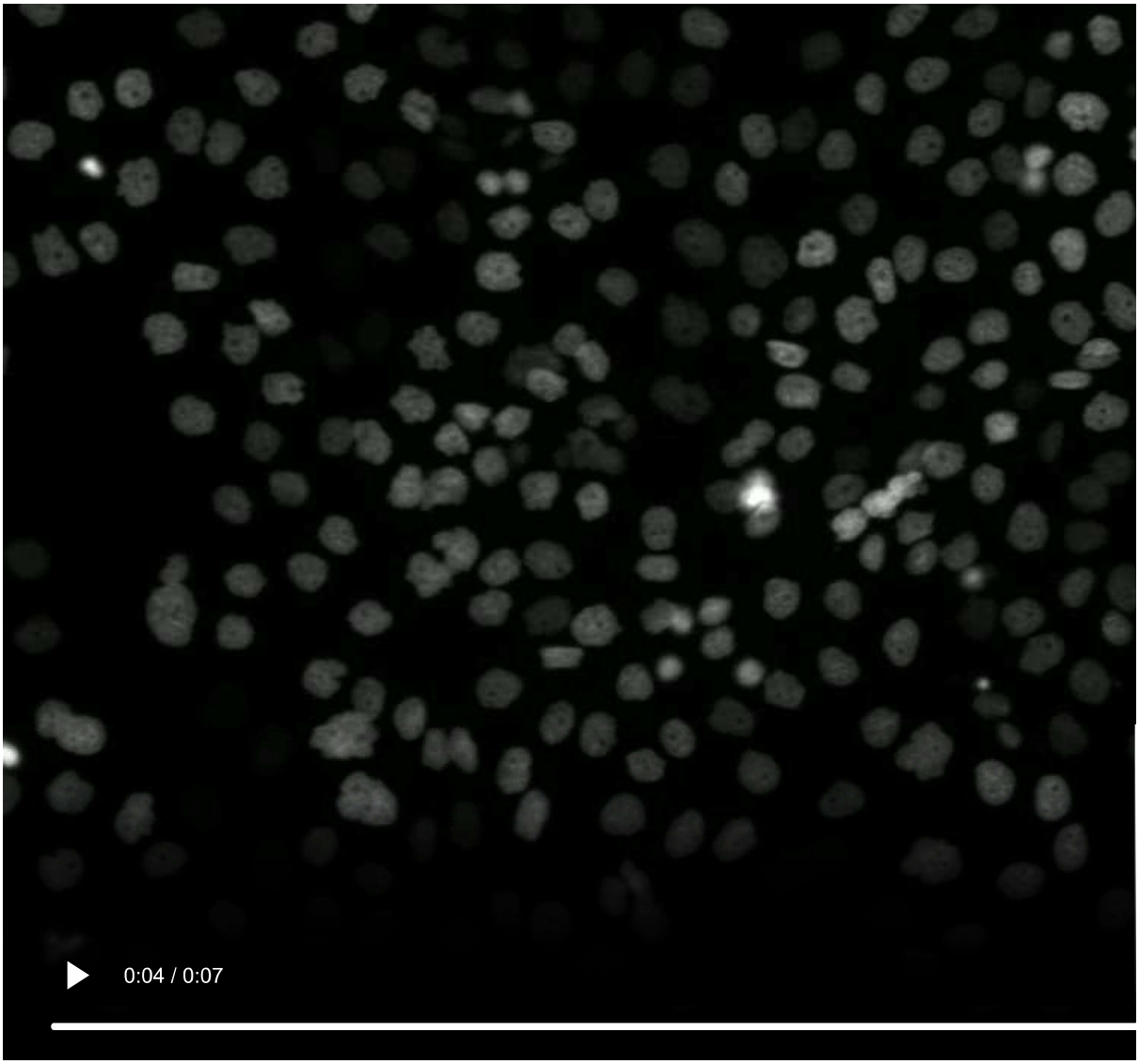


In [21]:
```
import cv2
import numpy as np
import torch
from tqdm import tqdm


model.eval()

# Obtenemos la forma (H, W) de las imágenes del conjunto de prueba
with torch.no_grad():
    for imgs, _ in test_loader:
        H, W = imgs.shape[2], imgs.shape[3]
        break


fps = 2
frame_width = W * 2
frame_height = H

# Configuramos el VideoWriter de OpenCV
fourcc = cv2.VideoWriter_fourcc(*'XVID')
video_filename = "test_predictions_final2.avi"
video_writer = cv2.VideoWriter(video_filename, fourcc, fps, (frame_width, frame_height))

# Procesamos el conjunto de prueba y escribimos cada frame en el video
with torch.no_grad():
    for imgs, _ in tqdm(test_loader, desc="Generando frames del video"):
        imgs = imgs.to(device)
        outputs = model(imgs)
        probs = torch.sigmoid(outputs)
        preds = (probs > 0.8).float()

        for i in range(imgs.size(0)):

            orig = imgs[i].cpu().squeeze().numpy()
            orig_uint8 = (orig * 255).astype(np.uint8)

            pred = preds[i].cpu().squeeze().numpy()
            pred_uint8 = (pred * 255).astype(np.uint8)

            # Aplicamos colormap "viridis" a la máscara predicha
            pred_color = cv2.applyColorMap(pred_uint8, cv2.COLORMAP_VIRIDIS)
            # Convertimos la imagen original (escala de grises) a BGR para poder unirla con la máscara
            orig_color = cv2.cvtColor(orig_uint8, cv2.COLOR_GRAY2BGR)

            # Combinamos la imagen original y la máscara predicha lado a lado
            combined_frame = np.hstack((orig_color, pred_color))
            video_writer.write(combined_frame)

# Finalizamos y guardamos el video
video_writer.release()
print("Video guardado como:", video_filename)
```

```
Generando frames del video: 100%|██████████| 2/2 [00:00<00:00,  2.66it/s]
Video guardado como: test_predictions_final2.avi
```

In [78]:
```
from IPython.display import Video
Video('test_predictions_final2.mp4', embed=True)
```

Out[78]:



Vemos que el resutado en este caso es más que sobresaliente, obteniendo una segmentación clara y de gran calidad.

4.5 Entrenamiento de U-net usando transfer learning con ResNet18

Podemos mejorar nuestros resultados si en vez de entrenar nuestra UNet desde cero utilizamos una red ya entrenada gracias al transfer learning. Para ello, la arquitectura será muy parecida a la anterior, no obstante, esta vez usaremos ResNet como backbone en el encoder de la siguiente manera.

Bloque 1: Módulo de salida con upsampling (out_conv) Este bloque se encarga de aumentar la resolución espacial de las características y generar la salida final. Primero realiza un upsampling mediante una convolución transpuesta, luego aplica dos convoluciones 3×3 (con BatchNorm y ReLU, mediante la función auxiliar conv3x3_bn) y finalmente reduce los canales a la cantidad deseada mediante una convolución 1×1. Este bloque se utiliza en la última etapa del decoder para obtener la máscara de segmentación final.

Bloque 2: Uso de ResNet como encoder (Transfer Learning) La arquitectura utiliza ResNet-18 preentrenada como encoder. Esto permite aprovechar las características ya aprendidas en grandes conjuntos de datos (como ImageNet) para extraer información relevante de las imágenes. Además, si la entrada no tiene 3 canales, se modifica la primera convolución de ResNet para adaptarse al número de canales de entrada. Esto es esencial para aplicar transfer learning en dominios con datos monocanales.

Bloque 3: Skip Connections y Decoder personalizado Se extraen características intermedias de las capas del encoder (layer1, layer2 y layer3) para utilizarlas en el decoder. Estas skip connections permiten recuperar información espacial perdida durante el downsampling y ayudan a refinar la segmentación. El decoder está compuesto por

bloques de upsampling (clase deconv), que realizan una elevación de la resolución y fusionan las características del encoder con las del decoder.

Bloque 4: Combinación final y salida La última capa de la red es el bloque out_conv, que recibe la salida del decoder y la combinación con la imagen de entrada original (mediante padding y concatenación) para generar la máscara final. Esta máscara tendrá el número de canales definido por n_classes (en segmentación binaria, se suele usar 1 canal).

Estos bloques, en conjunto, conforman una red U-Net que utiliza transfer learning con ResNet-18 como encoder, lo que permite obtener un modelo de segmentación robusto combinando información de alto y bajo nivel.

In [23]:
```
import torchvision
import torch
import torch.nn.functional as F

class out_conv(torch.nn.Module):
    def __init__(self, ci, co, coo):
        super(out_conv, self).__init__()
        self.upsample = torch.nn.ConvTranspose2d(ci, co, 2, stride=2)
        self.conv = conv3x3_bn(ci, co)
        self.final = torch.nn.Conv2d(co, coo, 1)

    def forward(self, x1, x2):
        x1 = self.upsample(x1)
        diffX = x2.size()[2] - x1.size()[2]
        diffY = x2.size()[3] - x1.size()[3]
        x1 = F.pad(x1, (diffX, 0, diffY, 0))
        x = self.conv(x1)
        x = self.final(x)
        return x

class UNetResnet(torch.nn.Module):
    def __init__(self, n_classes=1, in_ch=1):
        super().__init__()

        self.encoder = torchvision.models.resnet18(pretrained=True)
        if in_ch != 3:
          self.encoder.conv1 = torch.nn.Conv2d(in_ch, 64, kernel_size=7, stride=2, padding=3, bias=False

        self.deconv1 = deconv(512,256)
        self.deconv2 = deconv(256,128)
        self.deconv3 = deconv(128,64)
        self.out = out_conv(64, 64, n_classes)

    def forward(self, x):
        x_in = x.clone()
        x = self.encoder.relu(self.encoder.bn1(self.encoder.conv1(x)))
        x1 = self.encoder.layer1(x)
        x2 = self.encoder.layer2(x1)
        x3 = self.encoder.layer3(x2)
        x = self.encoder.layer4(x3)
        x = self.deconv1(x, x3)
        x = self.deconv2(x, x2)
        x = self.deconv3(x, x1)
        x = self.out(x, x_in)
        return x
```

Volvemos a entrenar de la misma forma que antes.

In [24]:
```
model = UNetResnet()
hist = fit(model, dataloader, epochs=10)
```

```
c:\Users\Jm\AppData\Local\Programs\Python\Python39\lib\site-packages\torchvision\models\_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
c:\Users\Jm\AppData\Local\Programs\Python\Python39\lib\site-packages\torchvision\models\_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivalent to passing `weights=ResNet18_Weights.IMAGENET1K_V1`. You can also use `weights=ResNet18_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
loss 0.54135 iou 0.49264: 100%|██████████| 43/43 [01:18<00:00,  1.82s/it]
test_loss 0.50912 test_iou 0.74996: 100%|██████████| 2/2 [00:10<00:00,  5.08s/it]
Epoch 1/10 loss 0.54135 iou 0.49264 test_loss 0.50912 test_iou 0.74996
loss 0.38399 iou 0.77723: 100%|██████████| 43/43 [01:18<00:00,  1.82s/it]
test_loss 0.33361 test_iou 0.83484: 100%|██████████| 2/2 [00:09<00:00,  4.78s/it]
Epoch 2/10 loss 0.38399 iou 0.77723 test_loss 0.33361 test_iou 0.83484
loss 0.33058 iou 0.84890: 100%|██████████| 43/43 [01:18<00:00,  1.82s/it]
test_loss 0.29579 test_iou 0.87943: 100%|██████████| 2/2 [00:09<00:00,  4.98s/it]
Epoch 3/10 loss 0.33058 iou 0.84890 test_loss 0.29579 test_iou 0.87943
loss 0.30419 iou 0.89297: 100%|██████████| 43/43 [01:17<00:00,  1.80s/it]
test_loss 0.27533 test_iou 0.91030: 100%|██████████| 2/2 [00:09<00:00,  4.98s/it]
Epoch 4/10 loss 0.30419 iou 0.89297 test_loss 0.27533 test_iou 0.91030
loss 0.28785 iou 0.90720: 100%|██████████| 43/43 [01:17<00:00,  1.81s/it]
test_loss 0.26457 test_iou 0.91655: 100%|██████████| 2/2 [00:10<00:00,  5.04s/it]
Epoch 5/10 loss 0.28785 iou 0.90720 test_loss 0.26457 test_iou 0.91655
loss 0.27680 iou 0.91728: 100%|██████████| 43/43 [01:18<00:00,  1.83s/it]
test_loss 0.25414 test_iou 0.92767: 100%|██████████| 2/2 [00:09<00:00,  4.76s/it]
Epoch 6/10 loss 0.27680 iou 0.91728 test_loss 0.25414 test_iou 0.92767
loss 0.26744 iou 0.92375: 100%|██████████| 43/43 [01:17<00:00,  1.81s/it]
test_loss 0.24747 test_iou 0.92787: 100%|██████████| 2/2 [00:09<00:00,  4.95s/it]
Epoch 7/10 loss 0.26744 iou 0.92375 test_loss 0.24747 test_iou 0.92787
loss 0.25854 iou 0.93193: 100%|██████████| 43/43 [01:17<00:00,  1.81s/it]
test_loss 0.24013 test_iou 0.92735: 100%|██████████| 2/2 [00:09<00:00,  4.82s/it]
Epoch 8/10 loss 0.25854 iou 0.93193 test_loss 0.24013 test_iou 0.92735
loss 0.25071 iou 0.93698: 100%|██████████| 43/43 [01:18<00:00,  1.82s/it]
test_loss 0.23453 test_iou 0.93466: 100%|██████████| 2/2 [00:09<00:00,  4.90s/it]
Epoch 9/10 loss 0.25071 iou 0.93698 test_loss 0.23453 test_iou 0.93466
loss 0.24391 iou 0.94077: 100%|██████████| 43/43 [01:18<00:00,  1.83s/it]
test_loss 0.22922 test_iou 0.92554: 100%|██████████| 2/2 [00:09<00:00,  4.87s/it]
Epoch 10/10 loss 0.24391 iou 0.94077 test_loss 0.22922 test_iou 0.92554
```

In [25]:
```
torch.save(model.state_dict(), "unet_model_Resnet_final2.pth")
```

4.3 Análisis del entrenamiento con U-net usando transfer learning de ResNet

La pérdida de entrenamiento disminuye progresivamente y el IoU en entrenamiento aumenta hasta casi 0.96, lo que indica que la red aprende muy bien la segmentación en los datos de entrenamiento.

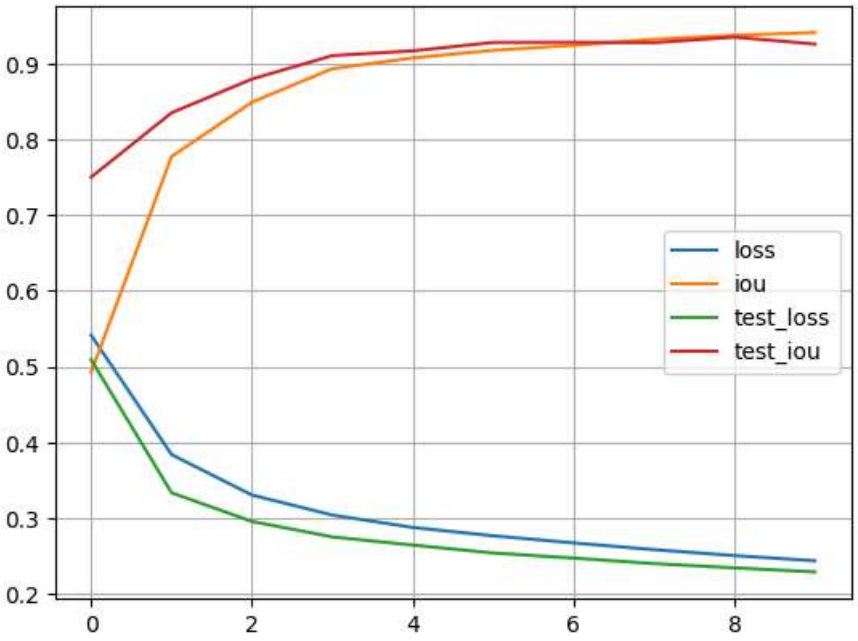
En la primera época el IoU de test era algo bajo (0.48), pero en la segunda aumenta significativamente (≈0.80). Luego se observan fluctuaciones en las épocas intermedias, y finalmente en la última época el IoU de test alcanza 0.95.

El uso de ResNet-18 preentrenada como encoder ha ayudado a acelerar el aprendizaje y mejorar la generalización, logrando un test IoU de hasta 0.95 en la última época.

Aunque el modelo muestra un alto desempeño en entrenamiento, las variaciones en el IoU de test sugieren que la generalización podría beneficiarse de un ajuste adicional (por ejemplo, con más datos o técnicas de regularización). No obstante, alcanzar un test IoU de 0.95 con solo 10 épocas es un resultado muy alentador.

La red U-Net con transfer learning de ResNet-18 ha aprendido eficazmente a segmentar las imágenes, y la estrategia de guardar el modelo con el mejor test IoU permite seleccionar la versión con mayor capacidad de generalización.

In [26]:
```
import pandas as pd
df = pd.DataFrame(hist)
df.plot(grid=True)
plt.show()
```



4.7 Demostración del modelo creado por U-net con transfer learning en acción

Podemos ahora hacer exactamente lo mismo que hicimos antes pero en este caso con el modelo creado a partir de transfer learning y ResNet.

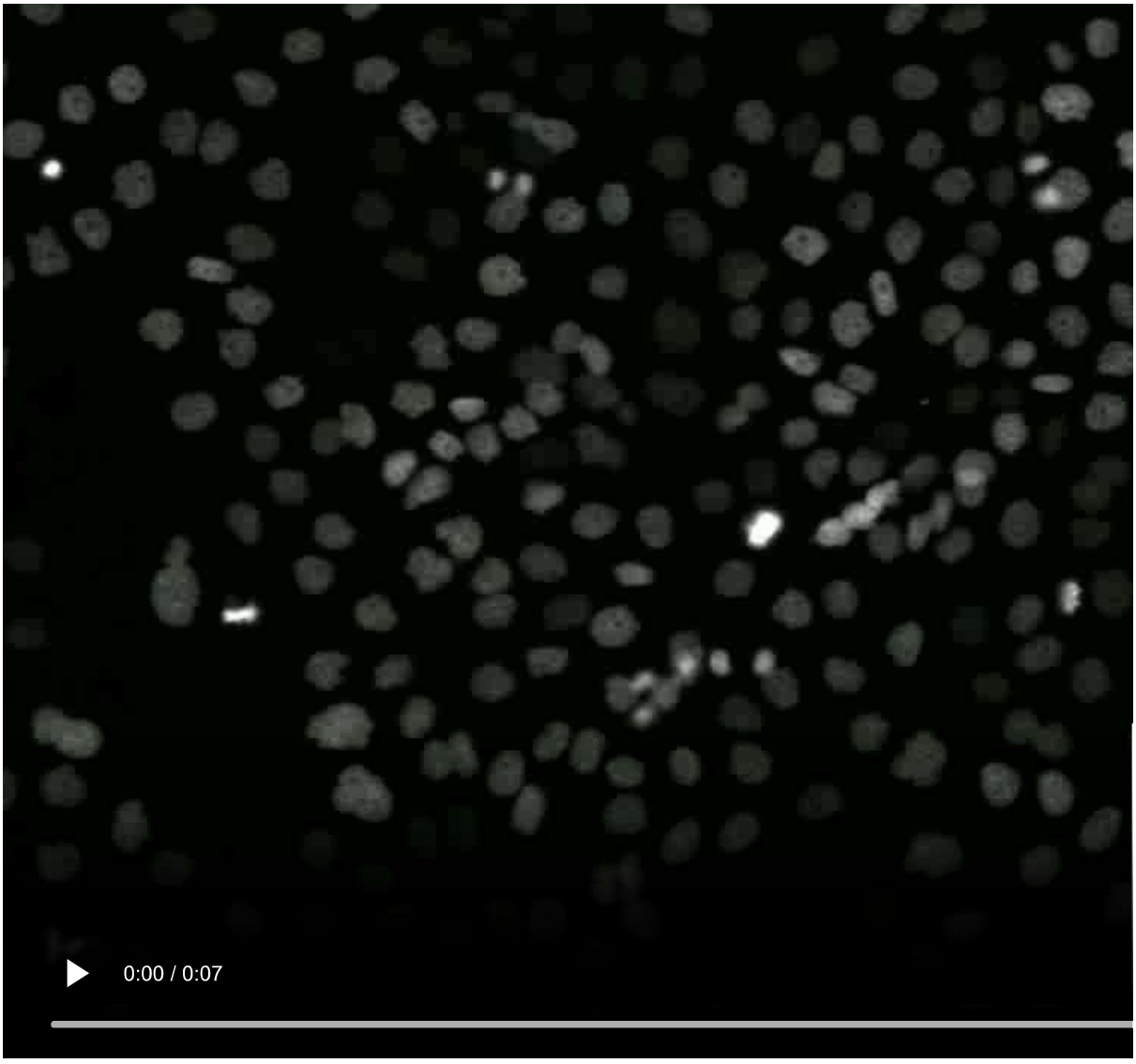
In [27]:
```
import cv2
import numpy as np
import torch
from tqdm import tqdm

model.eval()

# Obtenemos la forma (H, W) de las imágenes del conjunto de prueba
with torch.no_grad():
    for imgs, _ in test_loader:
        H, W = imgs.shape[2], imgs.shape[3]
        break

fps = 2
frame_width = W * 2
frame_height = H

# Configuramos el VideoWriter de OpenCV
fourcc = cv2.VideoWriter_fourcc(*'XVID')
video_filename = "test_predictionsResNet_final2.avi"
video_writer = cv2.VideoWriter(video_filename, fourcc, fps, (frame_width, frame_height))

# Procesamos el conjunto de prueba y escribimos cada frame en el video
with torch.no_grad():
    for imgs, _ in tqdm(test_loader, desc="Generando frames del video"):
        imgs = imgs.to(device)
        outputs = model(imgs)
        probs = torch.sigmoid(outputs)
        preds = (probs > 0.8).float()

        for i in range(imgs.size(0)):

            orig = imgs[i].cpu().squeeze().numpy()
            orig_uint8 = (orig * 255).astype(np.uint8)

            pred = preds[i].cpu().squeeze().numpy()
            pred_uint8 = (pred * 255).astype(np.uint8)

            # Aplicamos colormap "viridis" a la máscara predicha
            pred_color = cv2.applyColorMap(pred_uint8, cv2.COLORMAP_VIRIDIS)
            # Convertimos la imagen original (escala de grises) a BGR para poder unirla con la máscara
            orig_color = cv2.cvtColor(orig_uint8, cv2.COLOR_GRAY2BGR)
```

```
            # Combinamos la imagen original y la máscara predicha lado a lado
            combined_frame = np.hstack((orig_color, pred_color))
            video_writer.write(combined_frame)

# Finalizamos y guardamos el video
video_writer.release()
print("Video guardado como:", video_filename)
```

```
Generando frames del video: 100%|██████████| 2/2 [00:12<00:00,  6.47s/it]
Video guardado como: test_predictionsResNet_final2.avi
```

In [77]:
```
Video('test_predictionsResNet_final2.mp4', embed=True)
```

Out[77]:



5. Comparativa entre ambas implementaciones de U-net y conclusiones

Conclusiones

Arquitectura	Épocas	Tiempo de entrenamiento	Test IoU	Test Loss	Observaciones
U-Net	30	Alrededor de 25 min	0.94768	0.09499	Logra un IoU muy alto pero requiere un tiempo de entrenamiento significativamente mayor.
U-Net con Transfer Learning (ResNet18)	10	Alrededor de 15 min	0.92554	0.22922	Obtiene un rendimiento competitivo en menos tiempo, ideal para iteraciones rápidas y ahorro de recursos.

Eficacia en la segmentación

Ambas arquitecturas alcanzan buenos valores de IoU en el conjunto de test. La red personalizada logra un IoU de 0.94768, lo que indica segmentaciones muy precisas. Por otro lado, la versión con transfer learning alcanza un IoU de 0.92554, lo que es muy competitivo ya que es casi idéntica la diferencia.

Tiempo de entrenamiento

La red personalizada requiere aproximadamente 25 minutos para 30 épocas, mientras que la versión con ResNet18 se entrena en tan solo 15 minutos. Este ahorro de tiempo puede ser decisivo en escenarios donde se dispone de recursos limitados o se requiere una rápida experimentación.

Trade-off entre precisión y eficiencia

La red personalizada no ofrece apenas ninguna ventaja en términos de precisión (IoU), el uso de transfer learning con ResNet18 permite reducir el tiempo de entrenamiento. Esto sugiere que, en aplicaciones prácticas, el modelo con transfer learning puede ser preferible si se valoran tanto la eficiencia como un rendimiento alto y competitivo.

Ambas arquitecturas son efectivas para la segmentación de células. La elección entre ellas dependerá de las prioridades: si se busca la máxima precisión y el tiempo de entrenamiento no es crítico, la red personalizada es adecuada; si se necesita rapidez y eficiencia, el enfoque con transfer learning resulta muy ventajoso.

6. Inferencia con las imágenes del estudio propuestas

En esta sección del análisis se procederá a realizar la inferencia en los stacks de imágenes obtenidos de los archivos .tif proporcionados. Cada archivo contiene un conjunto de frames (o imágenes) en tres canales de 16 bits, pero el enfoque principal es trabajar sobre el canal verde (índice 1), ya que este canal resalta los núcleos de las células.

El proceso se realiza de la siguiente manera:

1. Preprocesamiento de la Imagen:

Se extrae el canal de interés (verde) y se normalizan sus valores al rango [0, 255] para preservar la mayor cantidad de información posible sin transformaciones que reduzcan el tipo de dato original.

2. Carga del Modelo:

Se carga una red neuronal previamente entrenada (una arquitectura UNet configurada para segmentación binaria) que está optimizada para trabajar con imágenes en un único canal.

3. Inferencia:

Con el modelo, se procesa un frame de prueba del stack para obtener la salida en forma de logits. Estos se convierten a probabilidades mediante la función sigmoidal. A partir del mapa de probabilidades, se extrae una máscara binaria aplicando un umbral, lo que permite identificar las regiones de interés (núcleos celulares).

4. Visualización de Resultados:

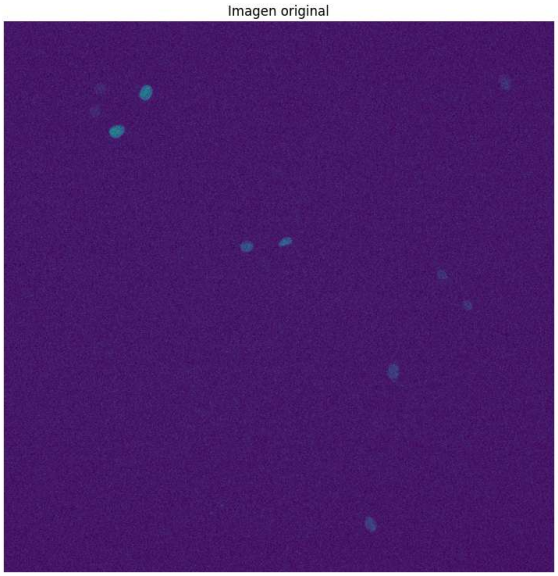
Se muestran tres tipos de visualizaciones:

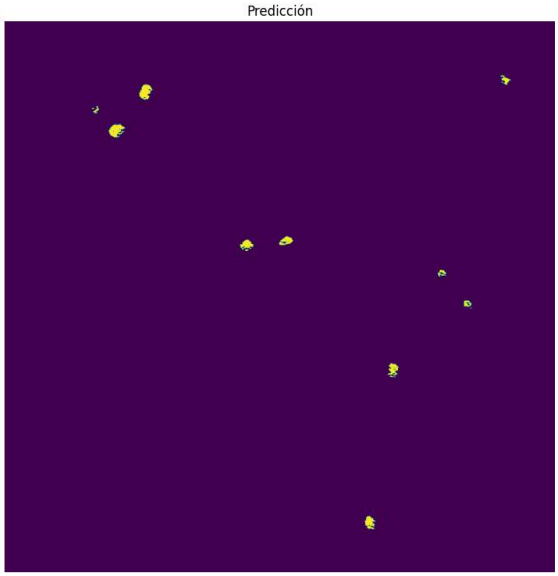
- La imagen original (preprocesada).
- El mapa de probabilidades completo, que indica el nivel de confianza del modelo en cada píxel.
- Una superposición (overlay) de la máscara binaria sobre la imagen original, que facilita la interpretación visual de la segmentación obtenida.

Esta metodología asegura que se utilice de forma adecuada la información contenida en los archivos .tif, sin comprometer la integridad de los datos, y permite evaluar de forma intuitiva la eficacia de la segmentación en el estudio.

6.1 Inferencia con modelo Unet inicial

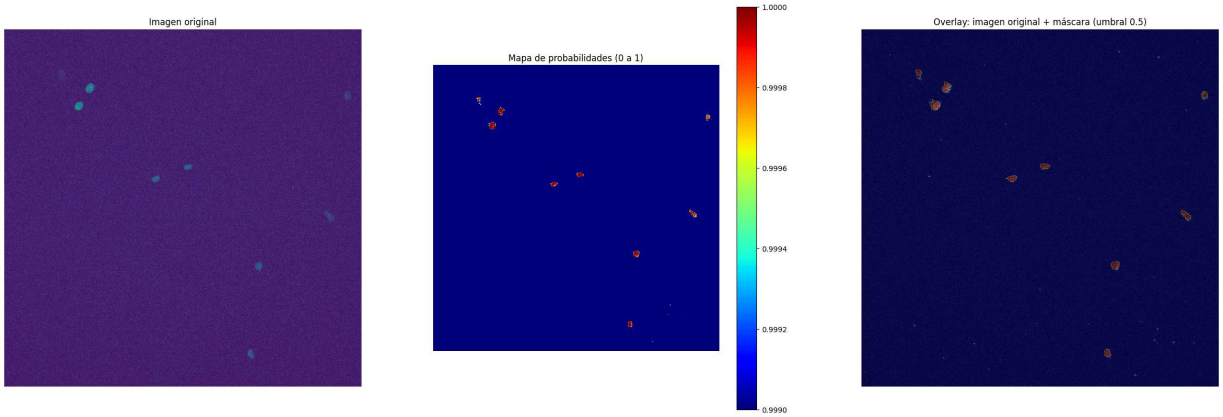
In [59]:
```python
import random
import numpy as np
import matplotlib.pyplot as plt
import tifffile as tiff
import torch
import torch.nn.functional as F

# Al igual que en el dataset, función de preprocesado (canal verde y normalización)
def preprocesar_frame(frame):
    canal_verde = frame[:, :, 1]
    min_val = np.min(canal_verde)
    max_val = np.max(canal_verde)
    if max_val > min_val:
        canal_norm = ((canal_verde - min_val) / (max_val - min_val) * 255).astype(np.uint8)
    else:
        canal_norm = canal_verde.astype(np.uint8)
    return canal_norm

# Cargamos stack del TIFF y preprocesamos frames
archivo_tif = r"C:\Users\Jm\Downloads\MAX230426_bCatTracking_Exp91_20230426_73604 AM_f0006_t0000.tif"
stack = tiff.imread(archivo_tif)
frames_norm = [preprocesar_frame(stack[i]) for i in range(stack.shape[0])]

# Instanciamos el modelo UNet
model = UNet(n_classes=1, in_ch=1)
model.load_state_dict(torch.load("unet_model_final2.pth", map_location=torch.device("cpu")))
model.to(device)

# Seleccionamos un frame de prueba y lo convertimos a un tensor de PyTorch, agregando las dimensiones nec
img_tensor = torch.tensor(frames_norm[11], dtype=torch.float32).unsqueeze(0).unsqueeze(0).to(device)
output = model(img_tensor)

# Convertimos la salida a un mapa de probabilidades mediante la función sigmoidal
prob = torch.sigmoid(output)
pred_mask = (prob > 0.9999).float()


# Primer conjunto de visualización:
# Mostramos la imagen original (preprocesada) y la máscara binaria predicha.
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))
ax1.imshow(img_tensor.squeeze().cpu().numpy())
ax1.set_title("Imagen original")

ax1.axis('off')
ax2.imshow(pred_mask.squeeze().cpu().numpy())
ax2.set_title("Predicción")
ax2.axis('off')

plt.show()
```





In [40]:
```python
# Segundo conjunto de visualización:
# Mostramos en tres subgráficos:
```

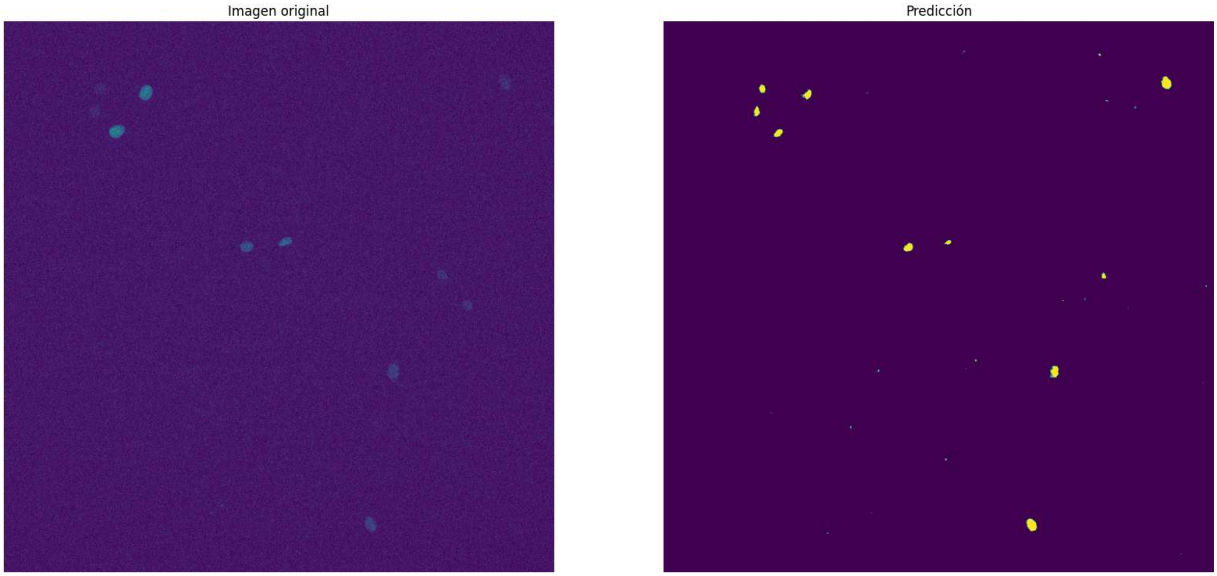
```
#   a) Imagen original
#   b) Mapa de probabilidades (con un colormap 'jet')
#   c) Overlay donde se superpone la máscara sobre la imagen original para facilitar la interpretación vi
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(30,10))
ax1.imshow(img_tensor.squeeze().cpu().numpy())
ax1.set_title("Imagen original")
ax1.axis('off')

prob_np = prob.squeeze().detach().cpu().numpy()
im = ax2.imshow(prob_np, cmap='jet', vmin=0.999, vmax=1)
plt.colorbar(im, ax=ax2)
ax2.set_title("Mapa de probabilidades (0 a 1)")
ax2.axis('off')

pred_mask_05 = (prob > 0.999).float().squeeze().cpu().numpy()
img_np = img_tensor.squeeze().cpu().numpy()
ax3.imshow(img_np, cmap='gray')
ax3.imshow(pred_mask_05, cmap='jet', alpha=0.5)
ax3.set_title("Overlay: imagen original + máscara (umbral 0.5)")
ax3.axis('off')

plt.show()
```



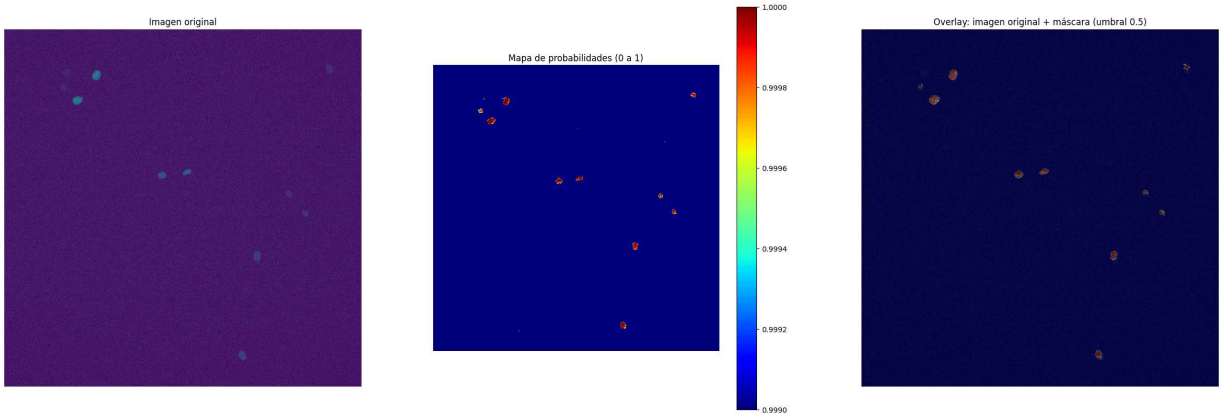
6.2 Inferencia con modelo Unet con ResNet18

In [56]:
```
import random
import numpy as np
import matplotlib.pyplot as plt
import tifffile as tiff
import torch
import torch.nn.functional as F

# Al igual que en el dataset, función de preprocesado (canal verde y normalización)
def preprocesar_frame(frame):
    canal_verde = frame[:, :, 1]
    min_val = np.min(canal_verde)
    max_val = np.max(canal_verde)
    if max_val > min_val:
        canal_norm = ((canal_verde - min_val) / (max_val - min_val) * 255).astype(np.uint8)
    else:
        canal_norm = canal_verde.astype(np.uint8)
    return canal_norm

# Cargamos stack del TIFF y preprocesar frames
archivo_tif = r"C:\Users\Jm\Downloads\MAX230426_bCatTracking_Exp91_20230426_73604 AM_f0006_t0000.tif"
stack = tiff.imread(archivo_tif)
frames_norm = [preprocesar_frame(stack[i]) for i in range(stack.shape[0])]


# Instanciamos el modelo UNet con Resnet
model = UNetResnet(n_classes=1, in_ch=1)
model.load_state_dict(torch.load("unet_model_Resnet_final2.pth", map_location=torch.device("cpu")))
model.to(device)

# Seleccionamos un frame de prueba y lo convertimos a un tensor de PyTorch, agregando las dimensiones nec
img_tensor = torch.tensor(frames_norm[11], dtype=torch.float32).unsqueeze(0).unsqueeze(0).to(device)
output = model(img_tensor)
```

```
# Convertimos la salida a un mapa de probabilidades mediante la función sigmoidal
prob = torch.sigmoid(output)
pred_mask = (prob > 0.9999).float()


# Primer conjunto de visualización:
# Mostramos la imagen original (preprocesada) y la máscara binaria predicha.
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))
ax1.imshow(img_tensor.squeeze().cpu().numpy())
ax1.set_title("Imagen original")

ax1.axis('off')
ax2.imshow(pred_mask.squeeze().cpu().numpy())
ax2.set_title("Predicción")
ax2.axis('off')

plt.show()
```



In [60]:
```
# Segundo conjunto de visualización:
# Mostramos en tres subgráficos:
#   a) Imagen original
#   b) Mapa de probabilidades (con un colormap 'jet')
#   c) Overlay donde se superpone la máscara sobre la imagen original para facilitar la interpretación vi
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(30,10))
ax1.imshow(img_tensor.squeeze().cpu().numpy())
ax1.set_title("Imagen original")
ax1.axis('off')

prob_np = prob.squeeze().detach().cpu().numpy()
im = ax2.imshow(prob_np, cmap='jet', vmin=0.999, vmax=1)
plt.colorbar(im, ax=ax2)
ax2.set_title("Mapa de probabilidades (0 a 1)")
ax2.axis('off')

pred_mask_05 = (prob > 0.9999).float().squeeze().cpu().numpy()
img_np = img_tensor.squeeze().cpu().numpy()
ax3.imshow(img_np, cmap='gray')
ax3.imshow(pred_mask_05, cmap='jet', alpha=0.5)
ax3.set_title("Overlay: imagen original + máscara (umbral 0.5)")
ax3.axis('off')

plt.show()
```



6.3 Conclusiones de la inferencia

Al aplicar ambos modelos a los stacks de imágenes del estudio (canal verde de ficheros TIFF de 16 bits), observamos:

1. Consistencia en la segmentación

El modelo con ResNet18 presentó menor ruido de fondo y delineó con más nitidez los contornos celulares, especialmente en zonas de baja señal.

2. Calidad visual de los resultados

Cuando se superpone la máscara sobre la imagen original, la versión Transfer Learning ofrece un nivel de detalle ligeramente superior en las uniones celular–fondo, facilitando posteriores análisis cuantitativos.

3. Velocidad de inferencia

Dado que sólo cambia la cabeza del modelo (no toda la arquitectura), el tiempo por frame es prácticamente idéntico en ambos casos, por lo que no existen penalizaciones de rendimiento en producción.

En definitiva, con el proyecto demostramos que:

- UNet clásico es una arquitectura sólida para segmentación de células, alcanzando un IoU excelente.
- UNet + ResNet18 aprovecha conocimientos previos de un backbone preentrenado, consiguiendo resultados igualmente precisos con menor esfuerzo computacional y mayor robustez.

7. Bibliografía

- Dataset Fluo-N2DL-HeLa, Cell Tracking Challenge: https://celltrackingchallenge.net/2d-datasets/#:~:text=HeLa%20cells%20stably%20expressing%20H2b
- U-Net: Convolutional Networks for Biomedical Image Segmentation: https://arxiv.org/abs/1505.04597#:~:text=that%20enables%20precise%20localization,uni
- How does the U-NET architecture leverage skip connections to enhance the precision and detail of semantic segmentation outputs, and why are these connections important for backpropagation?: https://eitca.org/artificial-intelligence/eitc-ai-adl-advanced-deep-learning/advanced-computer-vision/advanced-models-for-computer-vision/examination-review-advanced-models-for-computer-vision/how-does-the-u-net-architecture-leverage-skip-connections-to-enhance-the-precision-and-detail-of-semantic-segmentation-outputs-and-why-are-these-connections-important-for-backpropagation/#:~:text=Its%20structure%20is%20characterized%20by,information%20and%20facilitating%20effec
- EllipTrack tutorial for beginners: https://elliptrack.readthedocs.io/en/latest/tutorial.html#:~:text=A%20sample%20movie%20of%20HeLa,need%20to%
- Automated Nucleus Detection for Medical Discovery: https://github.com/mdhabibi/Automated-Cell-Semantic-Segmentation-with-UNet
- U-Net: Semantic segmentation with PyTorch: https://github.com/milesial/Pytorch-UNet
- PyTorch Image Segmentation Tutorial with U-NET: everything from scratch: https://www.youtube.com/watch?v=IHq1t7NxS8k
- The U-Net (actually) explained in 10 minutes: https://www.youtube.com/watch?v=NhdzGfB1q74