Detección zonas inundadas con Semantic Segmentation

Configuración del ambiente

He realizado un ambiente fácilmente reproducible y que me permite utilizar mi gpu con Docker, vscode y su extensión devcontainers. La configuración es el json que se muestra en la Figura 1.

Figura 1

Devcontainer con gpu

```
{
    "name": "Python 3",
    "image": "mcr.microsoft.com/devcontainers/python:1-3.
    12-bullseye",
    "customizations": {
        "vscode": {
            "extensions": [
                "ms-toolsai.jupyter"
            ]
        }
    },
    "runArgs": [
        "--gpus=all"
    ],
    "postCreateCommand": [
        "nvidia-smi"
    ]
}
```

Contenido del archivo Devcontainer.json para la definición del ambiente de desarrollo en un container.

El dataset

Para efectos prácticos, he utilizado un jupyter notebook para desarrollar la actividad, ya que guarda los outputs para luego mostrarlos fácilmente.

He automatizado la descarga del dataset con el script mostrado en la Figura 2.

Figura 2

Descargando el dataset

```
! mkdir -p ./dataset
! curl -L -o ./dataset/archive.zip https://www.kaggle.com/api/v1/datasets/download/
faizalkarim/flood-area-segmentation
! unzip -o ./dataset/archive.zip -d ./dataset
! rm ./dataset/archive.zip
```

Celda de jupyter notebook descargando el dataset.

Preparando el ambiente y la unet

Primero he instalado e importado las librerías necesarias para el ejercicio.

A continuación, he traído el código de la unet y me he asegurado que todas las librerías son importadas y el código python correctamente leído.

Cargando las imágenes y sus máscaras

He cargado las imágenes desde las carpetas Image y Mask del dataset, las he escalado a 100x100 y pasado a tensores, como muestra la Figura 3.

Figura 3

Cargando imágenes y máscaras

```
images = os.listdir('./dataset/Image/')
masks = os.listdir('./dataset/Mask/')

image_tensor = list()
mask_tensor = list()

for image in images:
    dd = PIL.Image.open(f'./dataset/Image/{image}')
    tt = torchvision.transforms.functional.pil_to_tensor(dd)
    tt = torchvision.transforms.functional.resize(tt, (100, 100))

    tt = tt[None, :, :, :]
    tt = torch.tensor(tt, dtype=torch.float) / 255.

    if tt.shape != (1, 3, 100, 100):
        continue

    mask = image.replace('.jpg', '.png')
    dd = PIL.Image.open(f'./dataset/Mask/{mask}')
    mm = torchvision.transforms.functional.pil_to_tensor(dd)
    mm = mm.repeat(3,1,1)
    mm = torchvision.transforms.functional.resize(mm, (100, 100))
    mm = mm[:1, :, :]

    mm = torch.tensor((mm > 0.).detach().numpy(), dtype=torch.long)
    mm = torch.nn.functional.one_hot(mm)
    mm = torch.permute(mm,(0, 3, 1, 2))
    mm = torch.tensor(mm, dtype=torch.float)

    image_tensor.append(tt)
    mask_tensor.append(mm)

image_tensor = torch.cat(image_tensor)
mask_tensor = torch.cat(mask_tensor)
```

Proceso de carga del dataset a tensores.

El entrenamiento

Paso siguiente, se instancia la unet, creo los DataLoader con batch 16, el optimizador, y listas para almacenar los loss e indices de jaccard.

Defino los epochs con el entrenamiento y agrego prints para ver el avance.

La Figura 4 muestra el código de lo anteriormente mencionado.

Figura 4

Entrenando la unet

```
unet = UNet(3, 2).to('cuda')

dataloader_train_image = torch.utils.data.DataLoader(image_tensor, batch_size=16)
dataloader_train_target = torch.utils.data.DataLoader(mask_tensor, batch_size=16)

optim = torch.optim.Adam(unet.parameters(), lr=0.01)
scheduler = StepLR(optim, step_size=1, gamma=0.5)
cross_entropy = torch.nn.CrossEntropyLoss()

loss_list = list()
jaccard_list = list()

for epoch in range(20):
    running_loss = 0.
    unet.train()

    jaccard_epoch = list()
    for image, target in zip(dataloader_train_image, dataloader_train_target):
        image = image.to('cuda')
        target = target.to('cuda')

        pred = unet(image)

        loss = cross_entropy(pred, target)

        running_loss += loss.item()

        loss.backward()
        optim.step()

    unet.eval()
    for image, target in zip(dataloader_train_image, dataloader_train_target):
        image = image.to('cuda')
        target = target.to('cuda')

        pred = unet(image)

        _, pred_unflatten = torch.max(pred, 1)
        _, target_unflatten = torch.max(target, 1)

        intersection = torch.sum(pred_unflatten == target_unflatten, dim=(1,2)) / 10000.
        jaccard_epoch.append(torch.mean(intersection).detach().to('cpu'))

    print(f'Epoch {epoch+1}',f'Loss: {running_loss}',f'Jaccard: {sum(jaccard_epoch) / len
    (jaccard_epoch)}')

    jaccard_list.append(sum(jaccard_epoch) / len(jaccard_epoch))

    loss_list.append(running_loss)

    scheduler.step()
```

Código del entrenamiento realizado.

Evaluación

El resultado del epoch 20 es Epoch 20 Loss: 10.25 y Jaccard: 0.76. Una mejor evaluación requiere dividir el dataset en al menos train y test, pero dado el alcance de la Memoria 3 y el tiempo personal limitado, lo dejaré para un futuro trabajo.

La Figura 5 muestra un plot de los losses e indices de jaccard durante el entrenamiento.

Figura 5

Historia del entrenamiento.

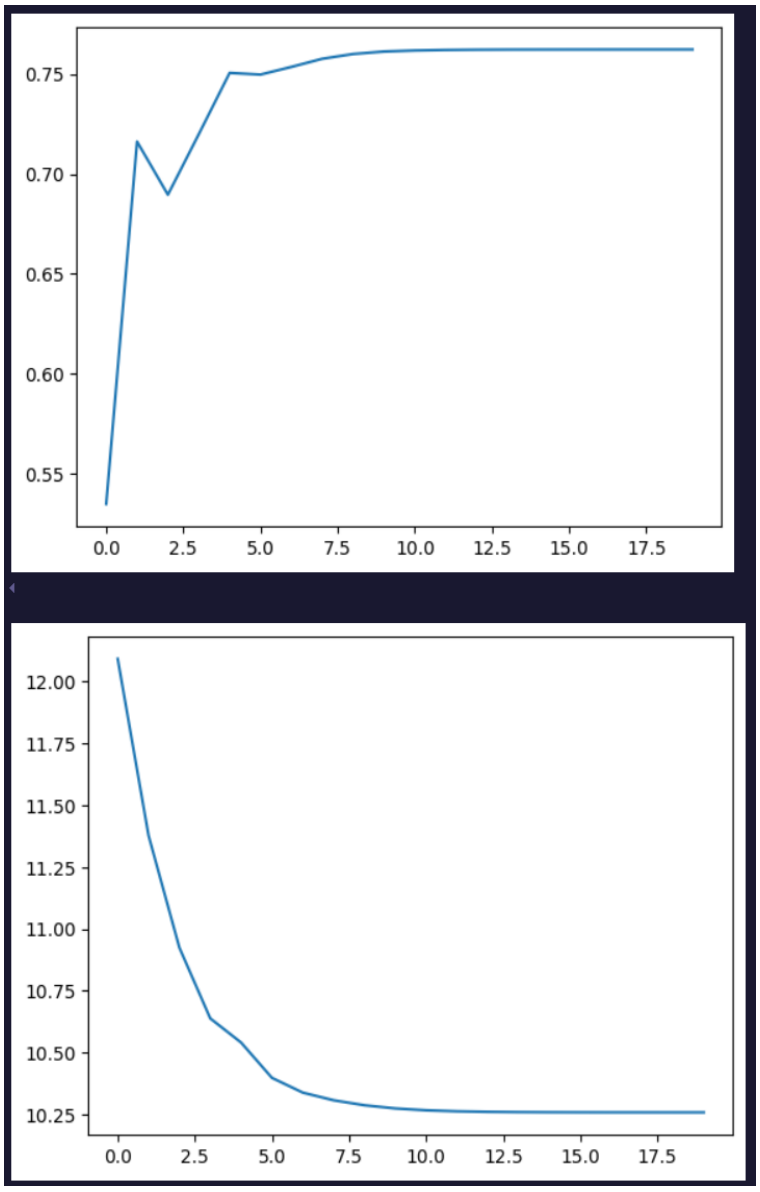

Grafico del losses e indices de jaccard del entrenamiento.

Una predicción

Finalmente, visualizo una predicción como se muestra en la Figura 6.

Figura 6

Predicción

Celda de jupyter notebook con el plot resultado de una predicción.