Computer Vision aplicado a diagnóstico de tumores

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 mostado en la Figura 2.

Figura 2

Downloading the dataset

Celda de jupyter notebook descargando el dataset.

Cargando las imágenes y sus clases

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

A continuación, creé un objeto Dataset para cargar y mapear las imágenes con sus respectivas clases, ya que estas se encuentran separadas en carpetas como benignos o malignos, como se muestra en la Figura 3.

Figura 3

SkinCancerDataset

```
class SkinCancerDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.transform = transform
        self.images = []
        self.labels = []
        self.classes = []
        # Iterate over the directories in the root directory
        for i, d in enumerate(os.listdir(root_dir)):
            self.classes.append(d)
            # Iterate over the files in each directory
            for f in os.listdir(os.path.join(root_dir, d)):
                self.images.append(os.path.join(root_dir, d, f))
                self.labels.append(i)

    def __len__(self):
        # Return the total number of images
        return len(self.images)

    def __getitem__(self, idx):
        # Load the image and its corresponding label
        image = Image.open(self.images[idx])
        label = self.labels[idx]
        # Apply the transformations if any
        if self.transform:
            image = self.transform(image)
        return image, label
```

Definición de la clase del Dataset.

Definí la transformación al tensor con previa escalada de los pixeles.

Como el dataset viene previamente dividido en train y test, he utilizado esta metodología para dividirlos. Más adelante divido nuevamente el dataset de entrenamiento en train y validation, para efectos de entramiento.

También he defino el DataLoader.

Figura 4

Carga del dataset

```python
# Define the transformation to be applied to the images
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor()
])

# Create the training and testing datasets
train_dataset = SkinCancerDataset('./dataset/train', transform=transform)
test_dataset = SkinCancerDataset('./dataset/test', transform=transform)

# Create the data loaders for training and testing datasets
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=32, shuffle=False)

print('Train dataset size:', len(train_dataset))
print('Train data samples:', train_dataset[0][0].shape)
print('Test dataset size:', len(test_dataset))
print('Test data samples:', test_dataset[0][0].shape)
```

✓ 0.0s Python

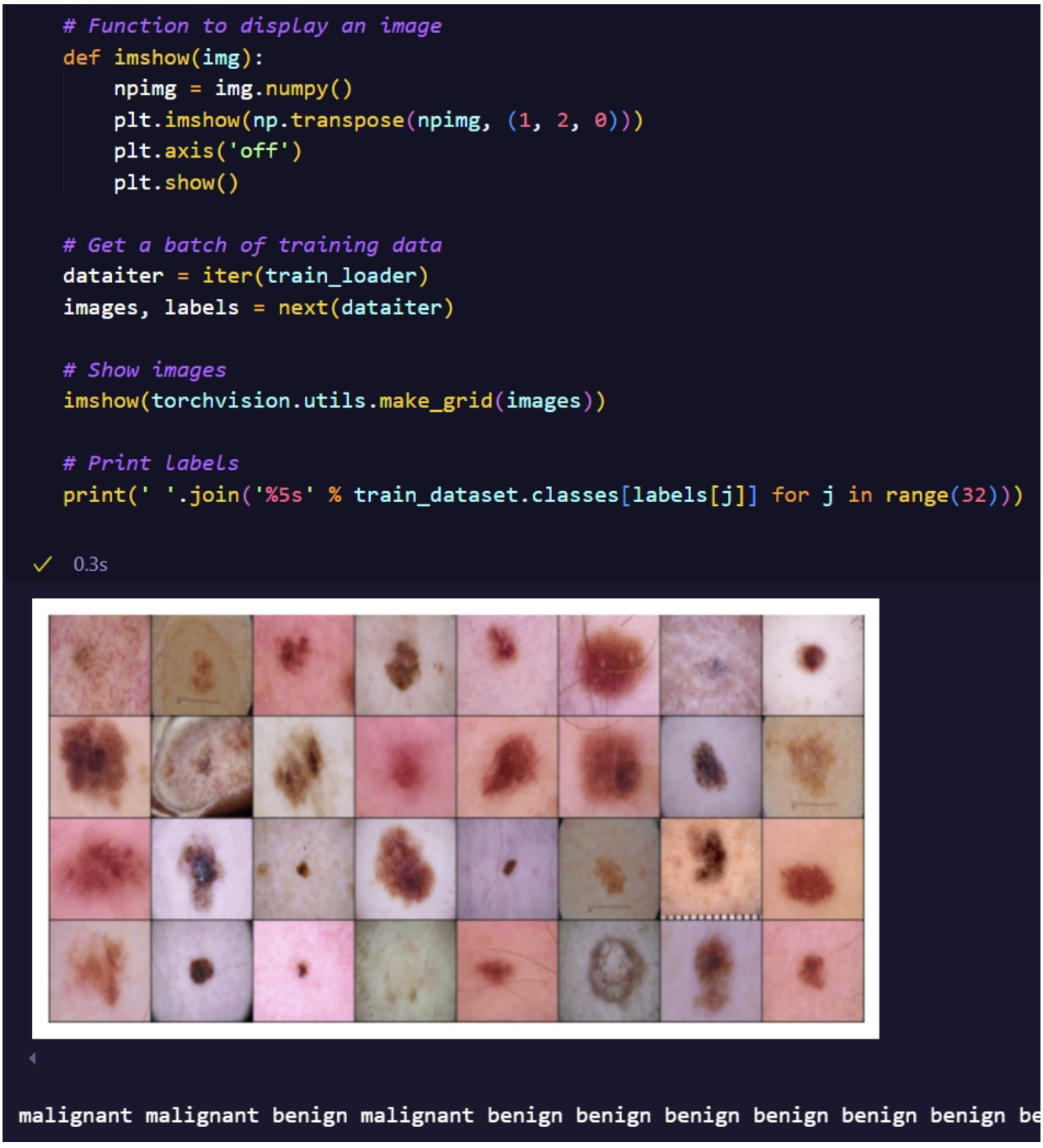
```
Train dataset size: 2637
Train data samples: torch.Size([3, 224, 224])
Test dataset size: 660
Test data samples: torch.Size([3, 224, 224])
```

Celda de jupyter notebook cargando el dataset en tensores.

Visualizamos las imágenes y sus labels en la Figura 5.

Figura 5

Visualizando



Celda de jupyter notebook mostrando algunos ejemplos del dataset.

Data augmentation

Paso siguiente, defino las transformaciones para hacer data augmentation como se muestra en la Figura 6. Esto solo lo aplico al dataset de entrenamiento.

Figura 6

Transformaciones

```python
# Define a class for random rotation
class RandomRotation:
    def __init__(self, degrees):
        self.degrees = degrees

    def __call__(self, x):
        angle = random.uniform(-self.degrees, self.degrees)
        return TF.rotate(x, angle)

# Define a class for random horizontal flip
class RandomHorizontalFlip:
    def __init__(self, p=0.5):
        self.p = p

    def __call__(self, x):
        if random.random() < self.p:
            return TF.hflip(x)
        return x

# Define a class for random vertical flip
class RandomVerticalFlip:
    def __init__(self, p=0.5):
        self.p = p

    def __call__(self, x):
        if random.random() < self.p:
            return TF.vflip(x)
        return x

# Define a class for random resized crop
class RandomResizedCrop:
    def __init__(self, size, scale=(0.5, 1.0), ratio=(0.75, 1.3333333333333333)):
        self.size = size
        self.scale = scale
        self.ratio = ratio

    def __call__(self, x):
        i, j, h, w = transforms.RandomResizedCrop.get_params(x, scale=self.scale,
        ratio=self.ratio)
        return TF.resized_crop(x, i, j, h, w, self.size)
```

Clases de python que definen transformaciones.

Las apliqué con un factor de aumento de 4, y creé un nuevo Dataset con la nueva data aumentada.

Figura 7

Data augmentation

```python
# Define the augmentation transform
transform_augmented = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToPILImage(),
    RandomRotation(10),
    RandomHorizontalFlip(),
    RandomVerticalFlip(),
    RandomResizedCrop((224, 224)),
    transforms.ToTensor()
])

# Create a new dataset with augmented samples
class AugmentedSkinCancerDataset(Dataset):
    def __init__(self, original_dataset, transform=None, augment_factor=4)
    :
        self.original_dataset = original_dataset
        self.transform = transform
        self.augment_factor = augment_factor

    def __len__(self):
        return len(self.original_dataset) * self.augment_factor

    def __getitem__(self, idx):
        original_idx = idx % len(self.original_dataset)
        image, label = self.original_dataset[original_idx]
        if self.transform:
            image = self.transform(image)
        return image, label

# Create the augmented dataset and dataloader
train_dataset_augmented = AugmentedSkinCancerDataset(train_dataset,
transform=transform_augmented)

print('Augmented train dataset size:', len(train_dataset_augmented))
```

```
✓ 0.0s                                                    Python
Augmented train dataset size: 10548
```

Celda de jupyter notebook aplicando data augmentation.

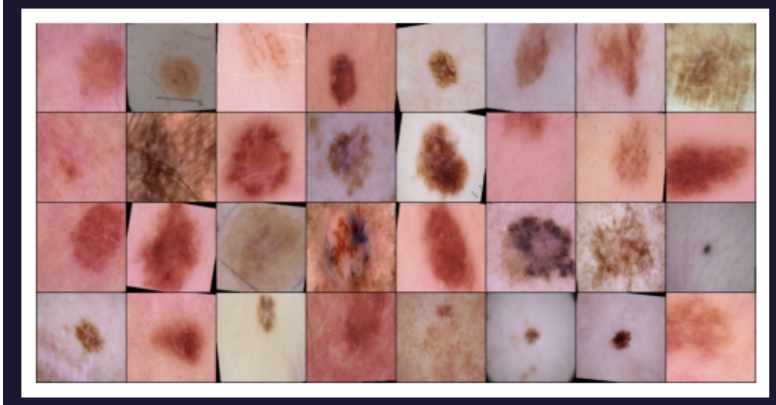
Dividí el dataset en train y validation. Visualizamos las nuevas imágenes que se muestran en la Figura 8.

Figura 8

Visualizando aumentada

```
# Split the augmented dataset into a training and validation set
train_dataset_augmented, val_dataset_augmented = torch.utils.data.random_split(
    train_dataset_augmented,
    [int(0.8 * len(train_dataset_augmented)), len(train_dataset_augmented) - int(0.8 * len
    (train_dataset_augmented))]
)

# Create data loaders for the augmented training and validation datasets
train_loader_augmented = torch.utils.data.DataLoader(train_dataset_augmented, batch_size=32, shuffle=True)
val_loader_augmented = torch.utils.data.DataLoader(val_dataset_augmented, batch_size=32, shuffle=False)

# Get a batch of augmented training data
dataiter = iter(train_loader_augmented)
images, labels = next(dataiter)

# Show augmented images
imshow(torchvision.utils.make_grid(images))

# Print dataset sizes and sample shapes
print('Train dataset augmented size:', len(train_dataset_augmented))
print('Train data augmented samples:', train_dataset_augmented[0][0].shape)
print('Validation dataset augmented size:', len(val_dataset_augmented))
print('Validation data augmented samples:', val_dataset_augmented[0][0].shape)
```

✓ 0.5s



```
Train dataset augmented size: 8438
Train data augmented samples: torch.Size([3, 224, 224])
Validation dataset augmented size: 2110
Validation data augmented samples: torch.Size([3, 224, 224])
```

Celda de jupyter notebook visualizando las nuevas imágenes generadas.

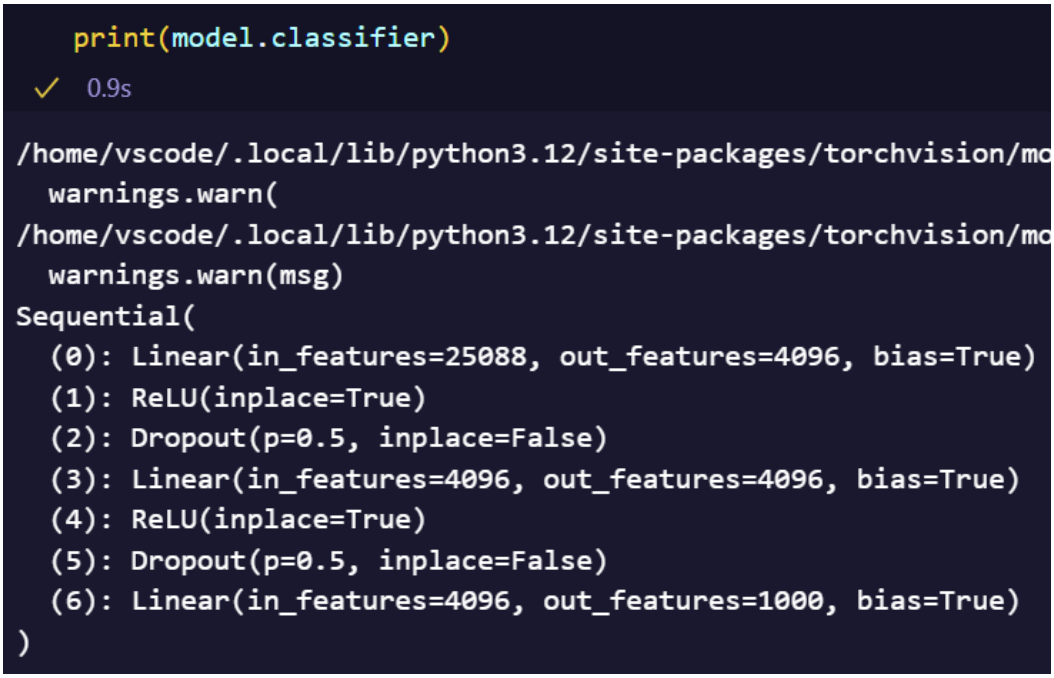
El entrenamiento

Paso siguiente, descargo e instancio el modelo vgg16 pre entrenado con la librería torchvision y su modelo models.vgg16(pretrained=True).

Luego visualizé las capas del clasificador que se aprecian en la Figura 9.

Figura 9

Capas del clasificador



Celda de jupyter notebook mostrando las capas del clasificador.

Reemplazo la cabeza por una nueva mostrada en la Figura 10.

Figura 10

Nueva cabeza

```
import torch.nn as nn

# Modify the classifier part of the VGG16 model
model.classifier = nn.Sequential(
    nn.Linear(25088, 4096),  # First fully connected layer
    nn.ReLU(inplace=True),   # ReLU activation
    nn.Dropout(0.5),         # Dropout layer
    nn.Linear(4096, 4096),   # Second fully connected layer
    nn.ReLU(inplace=True),   # ReLU activation
    nn.Dropout(0.5),         # Dropout layer
    nn.Linear(4096, 2)       # Output layer with 2 classes
)

# Print the modified classifier
print(model.classifier)
```

✓ 0.5s

```
Sequential(
  (0): Linear(in_features=25088, out_features=4096, bias=True)
  (1): ReLU(inplace=True)
  (2): Dropout(p=0.5, inplace=False)
  (3): Linear(in_features=4096, out_features=4096, bias=True)
  (4): ReLU(inplace=True)
  (5): Dropout(p=0.5, inplace=False)
  (6): Linear(in_features=4096, out_features=2, bias=True)
)
```

Celda de jupyter notebook definiendo la nueva cabeza.

Preparo las variables para entrenar el modelo. También creo las variables train_losses y train_accuracies para visualizarlas luego.

Ahora, defino el entrenamiento mostrado en la Figura 11.

Figura 11

El entrenamiento

```python
def train(model, train_loader, val_loader, criterion, optimizer, device, epochs=5):
    model.train()  # Set the model to training mode
    for epoch in range(epochs):
        print('Epoch', epoch + 1)
        running_loss = 0.0
        for i, data in enumerate(train_loader):
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()  # Zero the parameter gradients
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            if i % 10 == 9:  # Print every 10 mini-batches
                iter_loss = running_loss / 10
                train_losses.append(iter_loss)
                running_loss = 0.0
                _, predicted = torch.max(outputs, 1)
                correct = (predicted == labels).sum().item()
                accuracy = correct / labels.size(0)
                train_accuracies.append(accuracy)
                if i % 100 == 99:  # Print every 100 mini-batches
                    print('Iteration', i + 1, 'loss', iter_loss, 'accuracy', accuracy
        # Validation step
        model.eval()  # Set the model to evaluation mode
        val_loss = 0.0
        correct = 0
        total = 0
        with torch.no_grad():
            for data in val_loader:
                inputs, labels = data
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                _, predicted = torch.max(outputs, 1)
                correct += (predicted == labels).sum().item()
                total += labels.size(0)

        val_loss /= len(val_loader)
        val_accuracy = correct / total

        print('Epoch', epoch + 1, 'finished')
        print('Validation loss:', val_loss, 'accuracy:', val_accuracy)
        model.train()  # Set the model back to training mode
```

Definición del entrenamiento con Pytorch

Lo he corrido durante 5 epochs, dando los resultados de accuracy y lost para los datos de entrenamiento y validación mostrados en la Figura 12.

Figura 12

Entrenando

```
# Train the model for x epochs
train(model, train_loader_augmented, val_loader_augmented, criterion, optimizer, device, epochs=5)
```

✓ 20m 13.7s

```
Epoch 1
Iteration 100 loss 0.4785053670406342 accuracy 0.78125
Iteration 200 loss 0.3747273236513138 accuracy 0.8125
Epoch 1 finished
Validation loss: 0.3344055872523423 accuracy: 0.8464454976303317
Epoch 2
Iteration 100 loss 0.35457650721073153 accuracy 0.8125
Iteration 200 loss 0.3195867672562599 accuracy 0.8125
Epoch 2 finished
Validation loss: 0.29805411669341003 accuracy: 0.8568720379146919
Epoch 3
Iteration 100 loss 0.2524770855903625 accuracy 0.90625
Iteration 200 loss 0.3182057484984398 accuracy 0.8125
Epoch 3 finished
Validation loss: 0.2976756301341635 accuracy: 0.862085308056872
Epoch 4
Iteration 100 loss 0.24109092801809312 accuracy 0.9375
Iteration 200 loss 0.2205989859998226 accuracy 0.875
Epoch 4 finished
Validation loss: 0.2191461068436955 accuracy: 0.9033175355450237
Epoch 5
Iteration 100 loss 0.22011136636137962 accuracy 0.875
Iteration 200 loss 0.22743440717458724 accuracy 0.875
Epoch 5 finished
Validation loss: 0.26673601217793697 accuracy: 0.8796208530805687
```

Celda de jupyter notebook con el resultado del entrenamiento.

La historia del entrenamiento

La Figura 13 muestra un plot de los losses y accuracies de la fase de entrenamiento.

Figura 13

Historia del entrenamiento

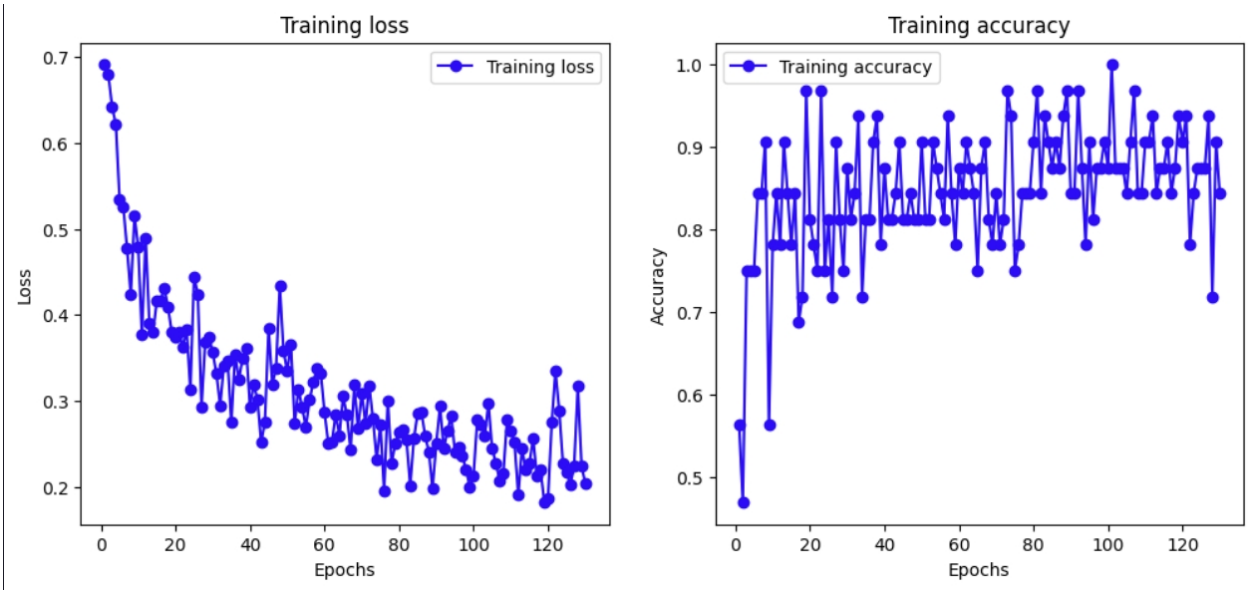


Grafico del losses y accuracies del entrenamiento.

La evaluación

Finalmente, evalúo el modelo con los 3 datasets (train, evaluation y test) con el código de la Figura 14.

Figura 14

Evaluación

```python
def evaluate(model, data_loader, device):
    model.eval()  # Set the model to evaluation mode
    loss = 0
    correct = 0
    total = 0
    with torch.no_grad():  # Disable gradient calculation
        for data in data_loader:
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss += criterion(outputs, labels).item()  # Accumulate the loss
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()  # Count correct predictions
    return loss / total, correct / total  # Return average loss and accuracy

# Evaluate the model on the training, validation, and test datasets
train_loss, train_accuracy = evaluate(model, train_loader, device)
val_loss, val_accuracy = evaluate(model, val_loader_augmented, device)
test_loss, test_accuracy = evaluate(model, test_loader, device)

# Print the evaluation results
print('Train loss:', train_loss)
print('Train accuracy:', train_accuracy)
print('Validation loss:', val_loss)
print('Validation accuracy:', val_accuracy)
print('Test loss:', test_loss)
print('Test accuracy:', test_accuracy)
```

✓ 1m 44.2s

```
Train loss: 0.008081126035638592
Train accuracy: 0.8839590443686007
Validation loss: 0.008298914060349712
Validation accuracy: 0.8834123222748815
Test loss: 0.009701880528309354
Test accuracy: 0.8560606060606061
```

Celda de jupyter notebook con el resultado del loss y accuracy de los datasets de entrenamiento, validación y test.

El resultado con el dataset de prueba es 85% de accuraccy, mientras que el de train y validation dan un 88%. Esto puede sugerir un pequeño sobreajuste, el cual lo dejaré para un futuro trabajo dada la demostración del caso.