Modelo para detección de trauma abdominal con imagénes radiográficas empleando Redes Convolucionales

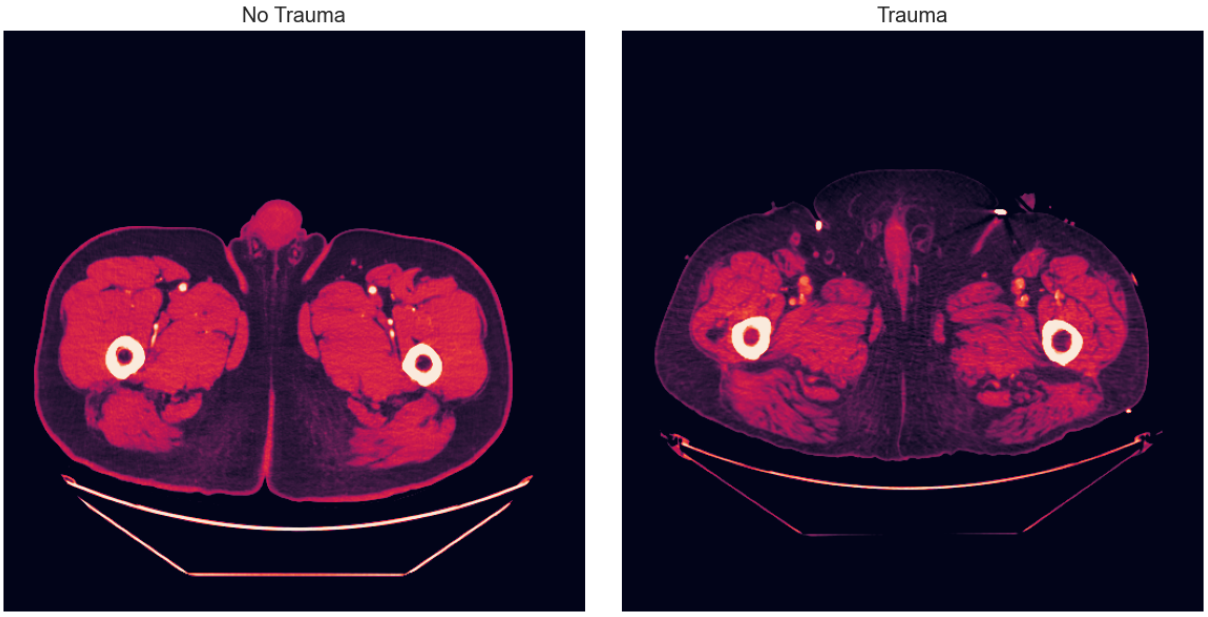
Experimentación con distintas arquitecturas de Redes Neuronales Convolucionales tanto diseñadas desde cero como aplicando técnicas de transfer learning

Desarrollo de competencia Kaggle: RSNA 2023 Abdominal Trauma Detection

https://www.kaggle.com/competitions/rsna-2023-abdominal-trauma-detection/overview

Problema a resolver: Desarrollo de modelo de AI que permita identificar a través de imágenes radiográficas si un paciente presente o no un trauma abdominal en etapa temprana

In [ ]:



Context: Blunt force abdominal trauma is among the most common types of traumatic injury, with the most frequent cause being motor vehicle accidents. Abdominal trauma may result in damage and internal bleeding of the internal organs, including the liver, spleen, kidneys, and bowel. Detection and classification of injuries are key to effective treatment and favorable outcomes. A large proportion of patients with abdominal trauma require urgent

surgery. Abdominal trauma often cannot be diagnosed clinically by physical exam, patient symptoms, or laboratory tests.

Importación de librerías

In [ ]:
```
# Operating System and File Management
import os
import shutil

# Medical Imaging
import pydicom
from PIL import Image

# Data Manipulation and Analysis
import pandas as pd
import numpy as np

# Machine Learning (PyTorch)
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import torch.optim as optim
import torch.nn.functional as F
from torch.cuda.amp import autocast, GradScaler

# Computer Vision (PyTorch)
import torchvision
from torchvision import transforms

# Data Splitting and Evaluation
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split

# Hyperparameter Tuning
import optuna
from optuna.trial import TrialState
from optuna.integration import SkoptSampler

# Data Visualization
import matplotlib.pyplot as plt
import seaborn as sns
from torch.utils.tensorboard import SummaryWriter

# Other
from ipywidgets import FileUpload, Output
import json
```

Procesamiento de datos

Descarga de datos del repositorio de la competencia Kaggle

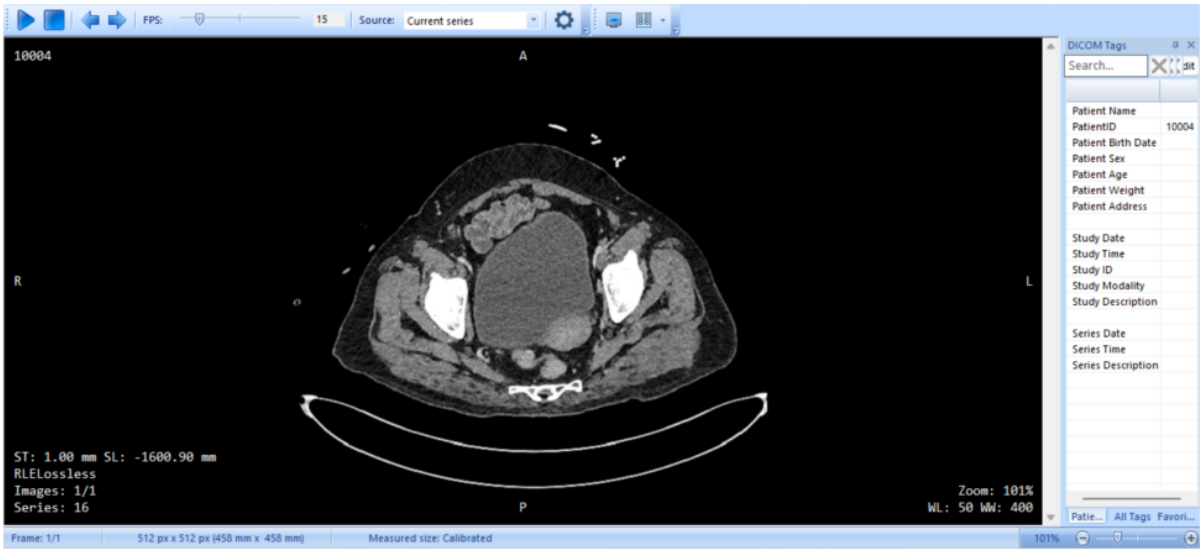
El siguiente código se encarga de descargar en un archivo comprimido el set de datos de Kaggle utilizando la API de la competencia: kaggle competitions download -c rsna-2023-abdominal-trauma-detection

In [ ]:
```
current_path = os.getcwd()
kaggle_json_path = os.path.join(current_path, "kaggle.json")

with open(kaggle_json_path) as f:
    kaggle_json_data = json.load(f)

os.environ["KAGGLE_USERNAME"] = kaggle_json_data["username"]
os.environ["KAGGLE_KEY"] = kaggle_json_data["key"]


def unzip_files():
    zip_file_path = "rsna-2023-abdominal-trauma-detection.zip"
    destination_folder = "images_dcm"
    shutil.unpack_archive(zip_file_path, destination_folder)
    return None

def main():
    !kaggle competitions download -c rsna-2023-abdominal-trauma-detection
    unzip_files()
    return None

if __name__ == "__main__":
    main()
```

Conversión de radiografías a formato PNG

Las imagénes médicas son descargadas en su formato original dicom (Digital Imaging and Communications in Medicine), por lo que para su procesamiento se deben pasar a un formato como .jpg o .png para aplicar las técnicas de ingeniería de imágenes y modelos de AI. El siguiente código convierte todas las imagenes descargadas en formato .dcm a un formato .png

In [ ]:
```
```



In [ ]:
```
def convertir_dicom_a_png(dir_dicom, dir_png):
    if not os.path.exists(carpetadir_png):
        os.makedirs(carpetadir_png)

    archivos_dicom = [archivo for archivo in os.listdir(carpetadir_dicom) if archiv

    for archivo in archivos_dicom:
        ruta_dicom = os.path.join(carpetadir_dicom, archivo)
        ds = pydicom.dcmread(ruta_dicom)
        nombre_png = archivo.replace('.dcm', '.png')
        ruta_png = os.path.join(carpetadir_png, nombre_png)
        plt.imsave(ruta_png, ds.pixel_array, cmap=plt.cm.bone)

carpetadir_dicom = 'images_dcm'  # Carpeta que contiene archivos DICOM
carpetadir_png = 'images_png'  # Carpeta de salida para las imágenes PNG
convertir_dicom_a_png(carpetadir_dicom, carpetadir_png)
```

Depuración de imagenes duplicadas

El siguiente código genera un repositorio con imágenes únicas, ya que muchas de las imágenes oringales se encuentran repetidas desde el repertorio original

In [ ]:
```
directorio_imagenes = "images_png"
directorio_destino = "images_uniques"

if not os.path.exists(directorio_destino):
    os.makedirs(directorio_destino)

ids_vistos = {}

for filename in os.listdir(directorio_imagenes):
    if filename.endswith(".png"):
        id_hash = filename.split("_")[0]

        if id_hash not in ids_vistos:
            nuevo_nombre = f"{id_hash}.png"
```

```
            shutil.copy(os.path.join(directorio_imagenes, filename), os.path.join(d
            ids_vistos[id_hash] = True
        else:
            os.remove(os.path.join(directorio_imagenes, filename))
```

Construcción del set de datos con la etiqueta de la imagen

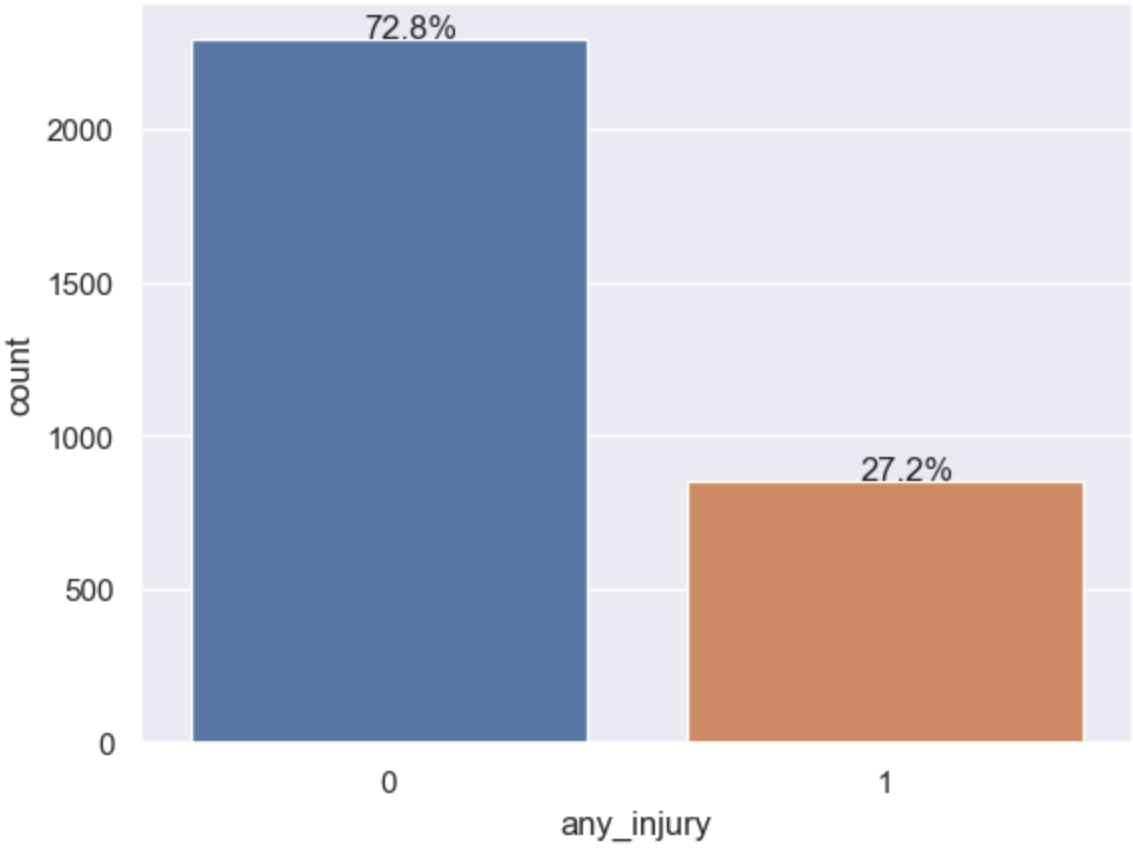
El siguiente código reparte las imagenes en dos repositorios, uno con imagenes que presenta un trauma (label 1) y otro con las imagenes que no presentan trauma (label 0), esto se realiza haciendo uso del csv 'target.csv', también descargado del repositorio de la competencia, el cual contiene los id de las imagenes con su respectiva etiquet

In [ ]:
```
image_folder = 'images_uniques'
folder_images_1 = 'images_1'
folder_images_0 = 'images_0'

csv_path = 'target.csv'
df = pd.read_csv(csv_path)

for index, row in df.iterrows():
    image_id = row['patient_id']
    target = row['any_injury']

    if target == 1:
        imagen_origen = os.path.join(image_folder, f'{image_id}.png')
        imagen_destino = os.path.join(folder_images_1, f'{image_id}.png')
        shutil.copy(imagen_origen, imagen_destino)
    else:
        imagen_origen = os.path.join(image_folder, f'{image_id}.png')
        imagen_destino = os.path.join(folder_images_0, f'{image_id}.png')
        shutil.copy(imagen_origen, imagen_destino)
```

Data Augmentation

El set de datos tiene un alto desbalanceo de clases, el 72,8% de las imagenes son de radiografías sin traumas y el 27,2% restante son imagenes con trauma.

In [ ]:
```
target_df = pd.read_csv('target/train.csv')
target_df['any_injury'].value_counts()
sns.set_theme(style="darkgrid")
ax = sns.countplot(x="any_injury", data=target_df)
total = len(target_df['any_injury'])
for p in ax.patches:
    percentage = '{:.1f}%'.format(100 * p.get_height()/total)
    x = p.get_x() + p.get_width() / 2 - 0.05
    y = p.get_y() + p.get_height() + 5
    ax.annotate(percentage, (x, y))
```



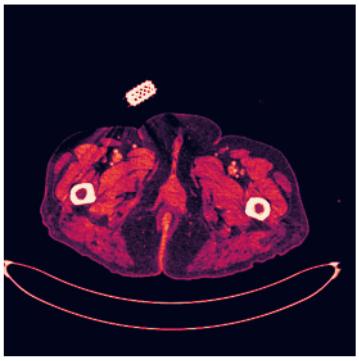
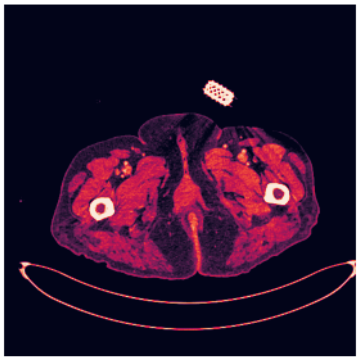
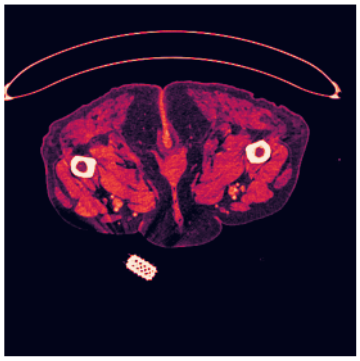
Con el objetivo de resolver el problema del desbalanceo de clases, se realiza un data augmentation generando el triple de imágenes pertenecientes a la clase 1 (imagenes con trauma). Para el data augmentation se utilizan dos técnicas: flip (voltear la imagen) y rotation 180 (rotación de 180 grados de la imagen)

In [ ]:
```
original_folder = 'images_1'
augmented_folder = 'augmented_images'
if not os.path.exists(augmented_folder):
    os.mkdir(augmented_folder)

image_files = [f for f in os.listdir(original_folder) if f.endswith('.png')]


for img_file in image_files:
    img_path = os.path.join(original_folder, img_file)
    img = Image.open(img_path)

    img.save(os.path.join(augmented_folder, img_file.split('.')[0] + '_original.png

    # Flip
    flipped_img = img.transpose(Image.FLIP_LEFT_RIGHT)
    flipped_img.save(os.path.join(augmented_folder, img_file.split('.')[0] + '_flip

    # Rotation
    rotated_img = img.rotate(180)
    rotated_img.save(os.path.join(augmented_folder, img_file.split('.')[0] + '_rota
```

```
print("Data augmentation complete.")
```

Ejemplo del resultado del data augmentation de la imagen original Imagen original, Imagen volteada, Imagen rotada 180

In [ ]:
```
sample_images = [f for f in os.listdir(augmented_folder) if f.endswith('.png')]
sample_images = sample_images[:3]
plt.figure(figsize=(12,12))
for i in range(0, len(sample_images)):
    plt.subplot(1, 3, i+1)
    image = Image.open(os.path.join(augmented_folder, sample_images[i]))
    plt.imshow(image)
    plt.axis('off')
```

El siguiente código genera el conjunto de datos final, el cual contiene todas las imagenes con su respectiva etiqueta

In [ ]:
```
directorio_imagenes_1 = "augmented_images"
directorio_imagenes_0 = "images_0"

archivos_1 = os.listdir(directorio_imagenes_1)
archivos_0 = os.listdir(directorio_imagenes_0)

data = [(os.path.join(directorio_imagenes_1, nombre), 1) for nombre in archivos_1]
       [(os.path.join(directorio_imagenes_0, nombre), 0) for nombre in archivos_0]
```

Ingeniería de imagenes

La variable transform es una variable que contiene una secuencia de métodos del objeto transforms de Pytorch para hacer ingeniería de imágenes, en este caso la imagen origial pasa por las siguientes transformaciones:

1. Se transforma a escala de grises (un único canal de color)
2. Se re dimensiona a 128x128 (las imágenes originales son de 560x560)
3. Se convierten las imagenes a Tensores (vectores númemricos con dimensión 128x128x1)
4. Se normalizan los tensores con media 0 y dsv 1. Esto se hace como método de escalado del valor de los pixeles ya que van desde 0 a 255

In [ ]:
```
transform = transforms.Compose([
    transforms.Grayscale(num_output_channels=1),
    transforms.Resize((128, 128)),
    transforms.ToTensor(),
    transforms.Normalize(mean=0., std=1.)
])
```

El dispositivo utilizado para el procesamiento de los datos y de las redes neuronales que se entrenarán es cuda aprovechando la ventaja de las GPU para el procesamiento en paralelo

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

Out[ ]:
```
device(type='cuda')
```

La siguiente clase CustomDataset contiene los métodos de construcción de los data loader tanto para los datos de entrenamiento como para los datos de validación. Los Data Loader generarán baches de 32 valores (tensores con su etiqueta)

In [ ]:
```
class CustomDataset(Dataset):
    def __init__(self, data, transform=None):
        self.data = data
        self.transform = transform

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

    def __getitem__(self, idx):
        img_name, label = self.data[idx]

        if os.path.exists(img_name):
            image = Image.open(img_name)
            label = torch.tensor(label, dtype=torch.float32)

            if self.transform:
                image = self.transform(image)

            return image, label
        else:
            return None, None

data_train, data_test = train_test_split(data, test_size=0.15, random_state=42, str

train_dataset = CustomDataset(data_train, transform=transform)
test_dataset = CustomDataset(data_test, transform=transform)

batch_size = 32

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

Red Neuronal Convolucional

Optimización de Hiper parámetros

La clase Net es la estructura de la red neuronal convolucional con hiperparámetros variables, los cuales son los que serán tuneados a través de la librería Optuna para encontrar la configuración óptima entre número de capas convolucionales, número de neuronas de las capas ocultas, tamaño de los filtros, optimizador, y learning rate

In [ ]:
```
class Net(nn.Module):
    def __init__(self, trial, num_conv_layers, num_filters, num_neurons, drop_conv2
        super(Net, self).__init__()
        in_size = 256
        kernel_size = 3

        self.convs = nn.ModuleList([nn.Conv2d(1, num_filters[0], kernel_size=(3, 3)
        out_size = in_size - kernel_size + 1
        out_size = int(out_size / 2)
        for i in range(1, num_conv_layers):
            self.convs.append(nn.Conv2d(in_channels=num_filters[i-1], out_channels=
            out_size = out_size - kernel_size + 1
            out_size = int(out_size/2)

        self.conv2_drop = nn.Dropout2d(p=drop_conv2)
        self.out_feature = num_filters[num_conv_layers-1] * out_size * out_size
        self.fc1 = nn.Linear(self.out_feature, num_neurons)
        self.fc2 = nn.Linear(num_neurons, 1)
        self.p1 = drop_fc1

        for i in range(1, num_conv_layers):
            nn.init.kaiming_normal_(self.convs[i].weight, nonlinearity='relu')
            if self.convs[i].bias is not None:
                nn.init.constant_(self.convs[i].bias, 0)
        nn.init.kaiming_normal_(self.fc1.weight, nonlinearity='relu')

    def forward(self, x):
        for i, conv_i in enumerate(self.convs):
            if i == 2:
                x = F.relu(F.max_pool2d(self.conv2_drop(conv_i(x)), 2))
            else:
                x = F.relu(F.max_pool2d(conv_i(x), 2))

        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, p=self.p1, training=self.training)
        x = F.sigmoid(self.fc2(x))
        return x

#La función de pérdida es Binary Cross Entropy al ser una tarea de clasificación bi
criterio = nn.BCEWithLogitsLoss()
os.environ["CUDA_LAUNCH_BLOCKING"] = "1"
```

Las siguientes son las funciones de entrenamiento y testeo de la CNN la cual utiliza el método GradScaler escalar los parámetros de la NN y agilizar así el entrenamiento.

In [ ]:
```
def train(model, optimizer):
    epochs = 10
    scaler = GradScaler()
    for epoch in range(epochs):
        model.train()
        total_loss = 0.0

        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad()

            with autocast():
                outputs = model(images)
                loss = criterio(outputs, labels.unsqueeze(1))

            scaler.scale(loss).backward()
            scaler.step(optimizer)
            scaler.update()
```

In [ ]:
```
def test(network):
    network.eval()
    correct = 0
    with torch.no_grad(), autocast():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = network(images)
            predicted = torch.round(torch.sigmoid(outputs))
            correct += (predicted == labels.unsqueeze(1)).sum().item()

    accuracy_test = correct / len(test_loader.dataset)

    return accuracy_test
```

La función objective es la encargada de realizar los ensayos con las distintas combinaciones de hiperparámetros para encontrar la mejor combinación, también se aprovecha el método de importance de la libería optuna para identificar los hiperparámetros con más relevancia sobre la función de error. La estructura de CNN que mejor rendimiento presente durante todos los ensayos se guardarán en un archivo csv llamado optuna_results.csv.

In [ ]:
```
def objective(trial):
    num_conv_layers = trial.suggest_int("num_conv_layers", 2, 3)
    num_filters = [int(trial.suggest_discrete_uniform("num_filter_"+str(i), 16, 128
                   for i in range(num_conv_layers)]
    num_neurons = trial.suggest_int("num_neurons", 10, 260, 10)
    drop_conv2 = trial.suggest_float("drop_conv2", 0.2, 0.5)
    drop_fc1 = trial.suggest_float("drop_fc1", 0.2, 0.5)

    model = Net(trial, num_conv_layers, num_filters, num_neurons, drop_conv2,  drop

    optimizer_name = trial.suggest_categorical("optimizer", ["Adam"])
    lr = trial.suggest_float("lr", 1e-5, 1e-2, log=True)
    optimizer = getattr(optim, optimizer_name)(model.parameters(), lr=lr)
```

```
    epochs = 10
    train(model, optimizer)
    accuracy = test(model)

    trial.report(accuracy, epochs)
    if trial.should_prune():
        raise optuna.exceptions.TrialPruned()

    return accuracy

if __name__ == '__main__':

    n_epochs = 12
    batch_size_train = 32
    batch_size_test = 32
    number_of_trials = 30
    limit_obs = True

    if limit_obs:
        number_of_train_examples = 2 * batch_size_train
        number_of_test_examples = 2 * batch_size_test
    else:
        number_of_train_examples = 32
        number_of_test_examples = 32

    random_seed = 1
    torch.backends.cudnn.enabled = False
    torch.manual_seed(random_seed)

    directory_name = 'files'
    if not os.path.exists(directory_name):
        os.mkdir(directory_name)

    sampler = SkoptSampler()
    study = optuna.create_study(direction="maximize", sampler = sampler)
    study.optimize(objective, n_trials=number_of_trials, n_jobs=-1)

    pruned_trials = study.get_trials(deepcopy=False, states=[TrialState.PRUNED])
    complete_trials = study.get_trials(deepcopy=False, states=[TrialState.COMPLETE]

    print("\nStudy statistics: ")
    print("  Number of finished trials: ", len(study.trials))
    print("  Number of pruned trials: ", len(pruned_trials))
    print("  Number of complete trials: ", len(complete_trials))

    trial = study.best_trial
    print("Best trial:")
    print("  Value: ", trial.value)
    print("  Params: ")
    for key, value in trial.params.items():
        print("    {}: {}".format(key, value))

    # Save results to csv file
    df = study.trials_dataframe().drop(['datetime_start', 'datetime_complete', 'dur
    df = df.loc[df['state'] == 'COMPLETE']
    df = df.drop('state', axis=1)
```

```
    df = df.sort_values('value')
    df.to_csv('optuna_results.csv', index=False)

    print("\nOverall Results (ordered by accuracy):\n {}".format(df))

    most_important_parameters = optuna.importance.get_param_importances(study, targ

    print('\nMost important hyperparameters:')
    for key, value in most_important_parameters.items():
        print('  {}:{}{:.2f}%'.format(key, (15-len(key))*' ', value*100))
```

Visualización de los Hiper parámetros del mejor resultado

In [ ]:
```
path = 'optuna_results.csv'
df = pd.read_csv(path)
df = df.sort_values('value', ascending=False)
best_trial = df.iloc[0].to_dict()
print(best_trial)
```

```
{'number': 21, 'value': 0.6666666666666666, 'params_drop_conv2': 0.352137010978494,
'params_drop_fc1': 0.2064119445870647, 'params_lr': 9.74180133852453e-05, 'params_nu
m_conv_layers': 3, 'params_num_filter_0': 64.0, 'params_num_filter_1': 96.0, 'params
_num_filter_2': 112.0, 'params_num_neurons': 260, 'params_optimizer': 'Adam'}
```

Al final de la ejecución de 30 ensayos se llega al resultado que la mejor combinación de la CNN es una red de 3 capas convolucionales con 64, 96, y 112 filtros cada una. Una capa oculta lineal con 260 neuronas, dos capas drop aut de regularización una con 0.35 de eliminación de neuronas aplicada a las capas convolucionales y otra de 0.20 aplicada a las capas lineales, y un optimizador Adam con un lr de 0.0000974.

Nota: Para la gran cantidad de posibles combinaciones que se pudiesen dar considerando el número de hiperparámetros a optimizar y el espacio de búsqueda tan amplio, son muy pocos ensayos para encontrar una combinación óptima global. Es recomendable ejecutar los ensayos utilizando una instancia con mayor capacidad de computo ya que localmente se tienen restricciones de memoria en la GPU para hacer una busqueda más exhaustiva de combinaciones. En el repositorio tengo un código con un pipeline de Sagemaker para esta tarea

CNN desde cero con hiper parámetros óptimos

Con los resultados anteriores, se construye la Red neuronal convolucional que con la estructura pseudo optimizada

In [ ]:
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()
```

```
        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.dropout_conv = nn.Dropout(0.2)
        self.dropout_fc = nn.Dropout(0.5)
        self.fc1 = nn.Linear(32768, 128)
        self.fc2 = nn.Linear(128, 1)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.conv2(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.conv3(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dropout_fc(x)
        x = self.fc2(x)
        return x

modelo = CNN().to(device)

criterio = nn.BCEWithLogitsLoss()
optimizador = optim.Adam(modelo.parameters(), lr=0.001)
```

In [ ]:
```
print(modelo)
```

```
CNN(
  (conv1): Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(64, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(96, 112, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (relu): ReLU()
  (sigmoid): Sigmoid()
  (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fal
se)
  (dropout_conv): Dropout(p=0.35, inplace=False)
  (dropout_fc): Dropout(p=0.2, inplace=False)
  (fc1): Linear(in_features=32768, out_features=260, bias=True)
  (fc2): Linear(in_features=260, out_features=1, bias=True)
)
```

A continuación se entrena la red neuronal con 20 epocas. Se instancia la clase SumaryWriter para cargar las ejecuciones a tensorboard y tener un mayor control del entrenamiento

In [ ]:
```
writer = SummaryWriter()
num_epochs = 20
losses = []
for epoch in range(num_epochs):
    modelo.train()
```

```
    total_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizador.zero_grad()
        outputs = modelo(images)
        loss = criterio(outputs, labels.unsqueeze(1))
        loss.backward()
        optimizador.step()
        total_loss += loss.item()

    epoch_loss = total_loss / len(train_loader)
    losses.append(epoch_loss)

    print(f'Época [{epoch + 1}/{num_epochs}], Pérdida: {total_loss / len(train_load
    writer.add_scalar('Loss/train', epoch_loss, epoch)

print('Training finished')
```

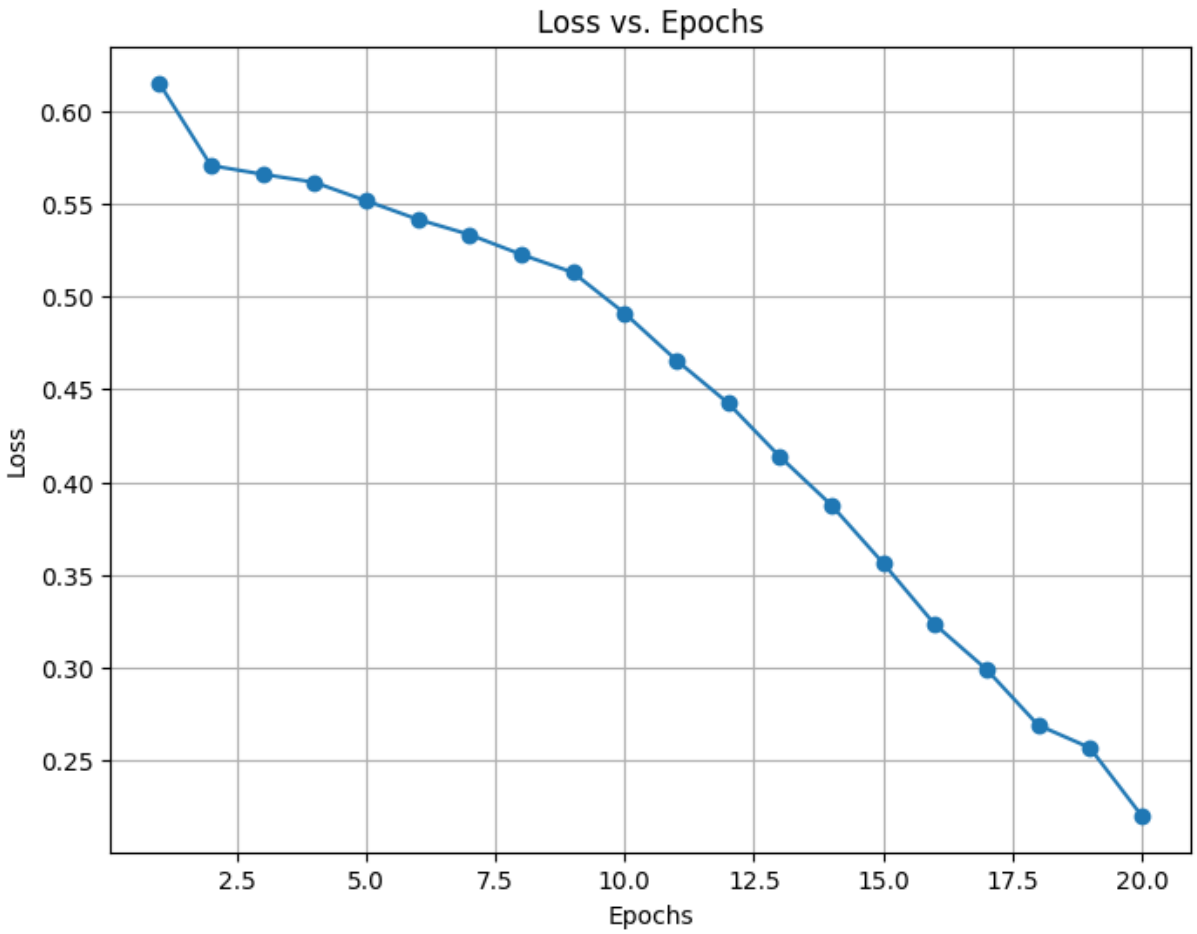
```
Época [1/20], Pérdida: 0.6049
Época [2/20], Pérdida: 0.5732
Época [3/20], Pérdida: 0.5665
Época [4/20], Pérdida: 0.5582
Época [5/20], Pérdida: 0.5534
Época [6/20], Pérdida: 0.5438
Época [7/20], Pérdida: 0.5308
Época [8/20], Pérdida: 0.5195
Época [9/20], Pérdida: 0.4989
Época [10/20], Pérdida: 0.4848
Época [11/20], Pérdida: 0.4600
Época [12/20], Pérdida: 0.4416
Época [13/20], Pérdida: 0.4153
Época [14/20], Pérdida: 0.3774
Época [15/20], Pérdida: 0.3662
Época [16/20], Pérdida: 0.3183
Época [17/20], Pérdida: 0.3133
Época [18/20], Pérdida: 0.2766
Época [19/20], Pérdida: 0.2728
Época [20/20], Pérdida: 0.2360
Training finished
```

Se agrega el modelo a la instancia de Tensorboard para su visualización gráfica

In [ ]:
```
writer.add_graph(modelo, images)
%load_ext tensorboard
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

In [ ]:
```
plt.figure(figsize=(8, 6))
plt.plot(range(1, num_epochs + 1), losses, marker='o', linestyle='-')
plt.title('Loss vs. Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.grid(True)
plt.show()
```



Se guarda el modelo como un objeto de PyTorch y se calcula la precisión para los datos de testeo utilizando el test_loader

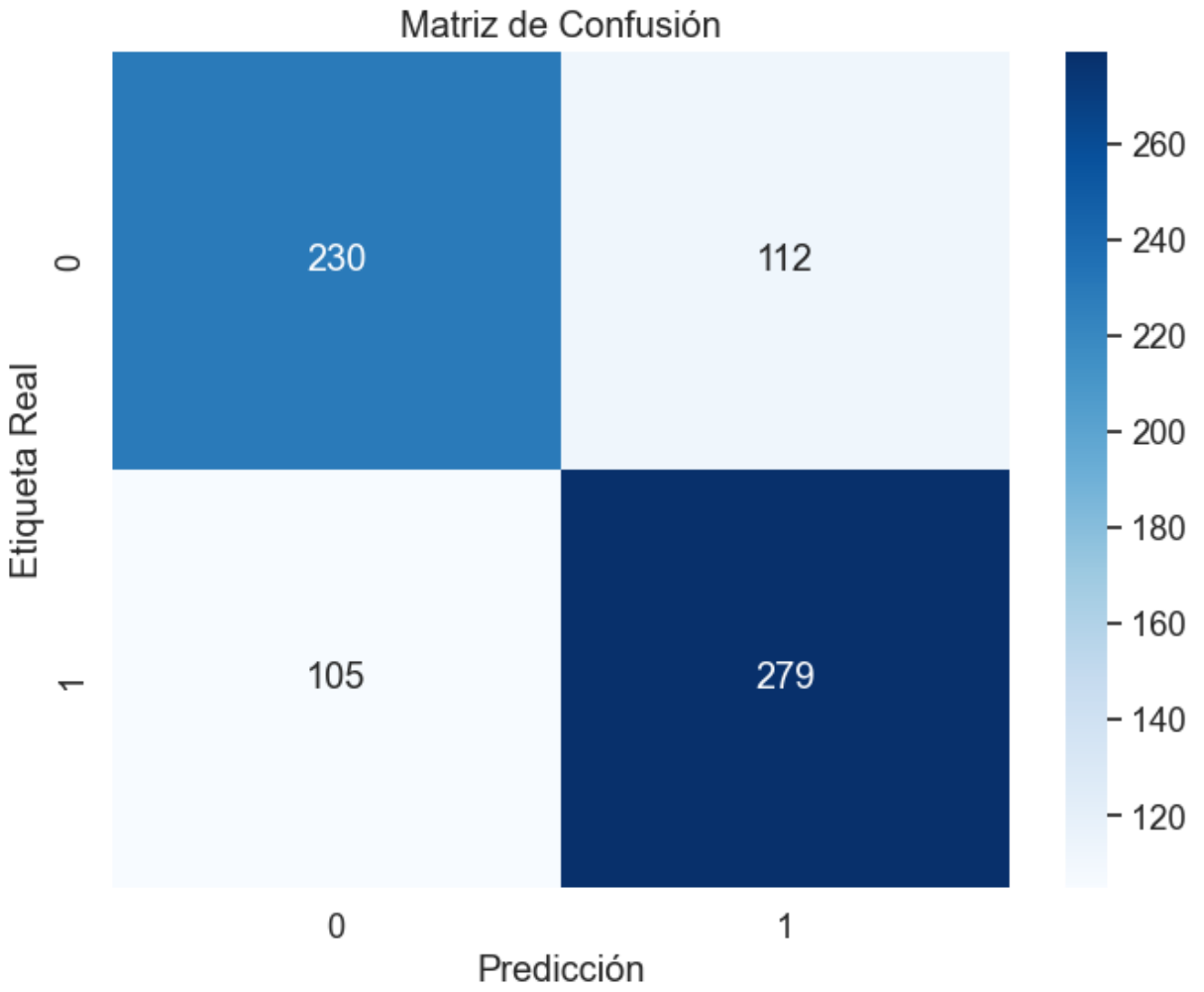
In [ ]:
```
torch.save(modelo.state_dict(), 'modelo.pth')
```

In [ ]:
```
modelo.eval()
total_correct = 0
total_samples = 0
with torch.no_grad():
    for data, targets in test_loader:
        data = data.to(device)
        targets = targets.to(device)
        outputs = modelo(data)
        targets = targets.unsqueeze(1).float()
        predictions = (outputs > 0.5).float()
        correct = (predictions == targets).sum().item()
        total_correct += correct
        total_samples += targets.size(0)
accuracy = (total_correct / total_samples) * 100.0
print(f'Accuracy on test set: {accuracy:.2f}%')
```

```
Accuracy on test set: 69.28%
```

In [ ]:
```
def plot_confusion_matrix(modelo, data_loader):
    class_names = ['0', '1']
    y_true = []
```

```
    y_pred = []
    modelo.eval()

    with torch.no_grad():
        for images, labels in data_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = modelo(images)
            predicted = (torch.sigmoid(outputs) > 0.5).float()
            y_true.extend(labels.cpu().numpy())
            y_pred.extend(predicted.cpu().numpy())

    # Calcular la matriz de confusión
    confusion = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(8, 6))
    sns.set(font_scale=1.2)  # Ajusta el tamaño de fuente para que sea legible
    sns.heatmap(confusion, annot=True, fmt="d", cmap="Blues", xticklabels=class_nam
    plt.xlabel("Predicción")
    plt.ylabel("Etiqueta Real")
    plt.title("Matriz de Confusión")
    plt.show()

plot_confusion_matrix(modelo, test_loader)
```



Feature visualization

Para aplicar la técnica de feature visualization, primero se guardan las capas convolucionales son sus respectivos pesos calculados durante el entrenamiento

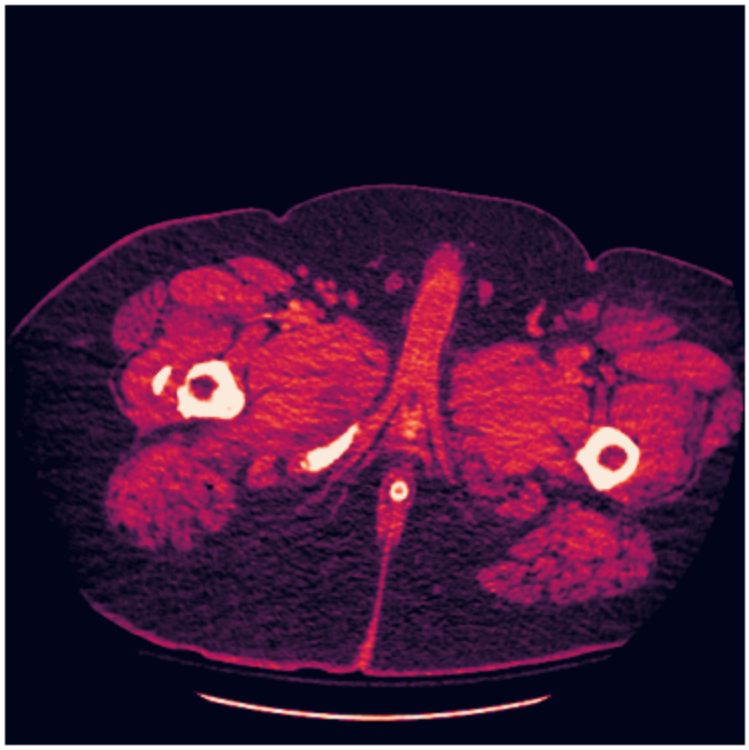
In [ ]:
```
model_weights =[]
conv_layers = []
model_children = list(modelo.children())
counter = 0
for i in range(len(model_children)):
    if type(model_children[i]) == nn.Conv2d:
        counter+=1
        model_weights.append(model_children[i].weight)
        conv_layers.append(model_children[i])
    elif type(model_children[i]) == nn.Sequential:
        for j in range(len(model_children[i])):
            for child in model_children[i][j].children():
                if type(child) == nn.Conv2d:
                    counter+=1
                    model_weights.append(child.weight)
                    conv_layers.append(child)
print(f"Total convolution layers: {counter}")
print(conv_layers)
```

```
Total convolution layers: 3
[Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)), Conv2d(32, 64, ke
rnel_size=(3, 3), stride=(1, 1), padding=(1, 1)), Conv2d(64, 128, kernel_size=(3,
3), stride=(1, 1), padding=(1, 1))]
```

A la siguiente imagen de prueba se le aplicarán los filtros convolucionales entrenados en la red neuronal para identificar el espacio latente de características resultantes al aplicarle las respectivas convoluciones

In [ ]:
```
image = Image.open(str('images_1/33.png'))
plt.imshow(image)
plt.axis('off')
```

Out[ ]:
```
(-0.5, 511.5, 511.5, -0.5)
```



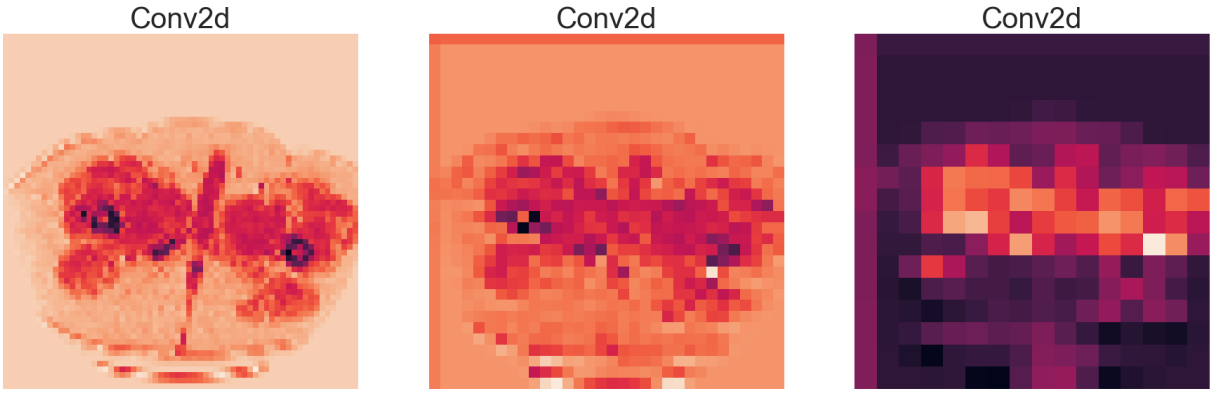
Se pasa la imagen por el transformador (canales a grises, re dimensionamiento, conversión a tensor, y normalización), y se aplican las 3 capas convolucionales para posteriormente guardar los vectores resultantes en un arreglo de valores que posteriormente serán graficados

In [ ]:
```
image = transform(image)
print(f"Image shape before: {image.shape}")
image = image.unsqueeze(0)
print(f"Image shape after: {image.shape}")
image = image.to(device)

outputs = []
names = []
processed = []
for layer in conv_layers[0:]:
    image = layer(image)
    outputs.append(image)
    names.append(str(layer))
print(len(outputs))
for feature_map in outputs:
    print(feature_map.shape)
for feature_map in outputs:
    feature_map = feature_map.squeeze(0)
    gray_scale = torch.sum(feature_map,0)
    gray_scale = gray_scale / feature_map.shape[0]
    processed.append(gray_scale.data.cpu().numpy())
```

```
Image shape before: torch.Size([1, 128, 128])
Image shape after: torch.Size([1, 1, 128, 128])
3
torch.Size([1, 32, 64, 64])
torch.Size([1, 64, 32, 32])
torch.Size([1, 128, 16, 16])
```

A continuación se muestra el espacio latente resultado de pasar la imagen por las 3 capas convolucionales. Estas imagenes muestran las regiones de activación sobre las que la red neuronal aprendió son más relevantes para la tarea de clasificación de trauma abdominal. Entre más profunda es la capa convolucional más abstracto es el patrón encontrado en la imagen por el filtro, lo que se puede traducir como una extracción de características de la imagen que va desde generalidades hasta particularidades que finalmente son aplanadas en un único vector que entra como input al clasificador que es un par de capas lineales con funciones de activación sigmoidales

In [ ]:
```
fig = plt.figure(figsize=(30, 50))
for i in range(len(processed)):
    a = fig.add_subplot(5, 4, i+1)
    imgplot = plt.imshow(processed[i])
    a.axis("off")
    a.set_title(names[i].split('(')[0], fontsize=30)
plt.savefig(str('feature_maps.jpg'), bbox_inches='tight')
```



Transfer Learning

Del repositorio de modelos de torchvision se importa la red convolucional entrada vgg19, la cual es una red muy profunda entrada con con 19 capas ocultas (16 convolucionales y 3 fully connected) y entrada con cientos de miles de imagenes de distintos objetos para una tarea de clasificación muticlase.

Esta red será tuneada para la tarea de clasificación de traumas abdominales, aprovechando toda su capacidad de extracción de características con sus filtros convolucionales ya entrenados, simplemente haciendo una modificación en la capa de entrada para que reciba imagenes de un solo canal de color y en la capa de salida para que genere una clasificación binaria

In [ ]:
```
from torchvision import models
model = models.vgg19(pretrained=True)
```

```
c:\AI\NN\dl\Lib\site-packages\torchvision\models\_utils.py:208: UserWarning: The par
ameter 'pretrained' is deprecated since 0.13 and may be removed in the future, pleas
e use 'weights' instead.
  warnings.warn(
c:\AI\NN\dl\Lib\site-packages\torchvision\models\_utils.py:223: UserWarning: Argumen
ts other than a weight enum or `None` for 'weights' are deprecated since 0.13 and ma
y be removed in the future. The current behavior is equivalent to passing `weights=V
GG19_Weights.IMAGENET1K_V1`. You can also use `weights=VGG19_Weights.DEFAULT` to get
the most up-to-date weights.
  warnings.warn(msg)
```

Estrcutura de la red neuronal vgg19

In [ ]:
```
model
```

Out[ ]:
```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fals
e)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fals
e)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace=True)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fals
e)
    (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (24): ReLU(inplace=True)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (26): ReLU(inplace=True)
    (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fals
e)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): ReLU(inplace=True)
    (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (33): ReLU(inplace=True)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (35): ReLU(inplace=True)
    (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fals
e)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): 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=1000, bias=True)
  )
)
```

A continuación se realiza la cutomización de la red neuronal vgg19 modificándose su capa de entrada para que reciba imagenes en escala de grises y su última capa para que genere resultados binarios como output

In [ ]:
```
for param in model.parameters():
    param.requires_grad = False
```

In [ ]:
```
model.features[0] = nn.Conv2d(1, 64, kernel_size=3, padding=1)
```

In [ ]:
```
for param in model.features[0].parameters():
    param.requires_grad = False
```

In [ ]:
```
model.classifier[-1] = nn.Sequential(
                       nn.Linear(4096, 260),
                       nn.ReLU(),
                       nn.Dropout(0.4),
                       nn.Linear(260, 1))
```

In [ ]:
```
model = model.to(device)
```

In [ ]:
```
num_epochs = 20
for epoch in range(num_epochs):
  model.train()
  total_loss = 0.0
  for data, targets in train_loader:
    data = data.to(device)
    targets = targets.to(device)
    optimizador.zero_grad()
    out = model(data)
    loss = criterio(out, targets.unsqueeze(1))
    loss.backward()
    optimizador.step()
    total_loss += loss.item()

print('Training finished')
```

```
Training finished
```

La red neuronal vgg19 no presentó buenos resultados, probablemente por el tipo de tarea e imagenes con la que fue entrenado que distan mucho de los patrones de las imágemes de trauma abdominal, sin embargo la falta de imagenes de entrenamiento y capacidad de computo también pueden generar un entrenammiento poco efectivo cuando se utilizan estructuras de redes neuronales tan profundas

Con base a lo mencionado anteriormente sobre la red vgg19, se decide utilizar una red neuronal pre entrenada especialmente con imagenes médicas como radiografías, la cual es la red ChexNet o su versión de torchvision densenet121.

A esta red neuronal se le aplica el mismo procedimiento para hacer el transfer learning, a esta red es necesario modificar las capas fully connected del final para modificar el resultado

de la salida y la capa de entrada para que reciba imagenes con un solo canal de color

In [ ]:
```
from torchvision import models
chexnet = models.densenet121(pretrained=True)
```

```
c:\AI\NN\dl\Lib\site-packages\torchvision\models\_utils.py:208: UserWarning: The par
ameter 'pretrained' is deprecated since 0.13 and may be removed in the future, pleas
e use 'weights' instead.
  warnings.warn(
c:\AI\NN\dl\Lib\site-packages\torchvision\models\_utils.py:223: UserWarning: Argumen
ts other than a weight enum or `None` for 'weights' are deprecated since 0.13 and ma
y be removed in the future. The current behavior is equivalent to passing `weights=D
enseNet121_Weights.IMAGENET1K_V1`. You can also use `weights=DenseNet121_Weights.DEF
AULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

Modificación de la capa lineal con una sola neurona a la salida de acuerdo a nuestra tarea de clasificación binaria

In [ ]:
```
chexnet.classifier = nn.Sequential(
    nn.Linear(1024, 1),
    nn.Sigmoid()
)
```

Modificación de la capa de entrada para que reciba imagenes con un solo canal de color

In [ ]:
```
chexnet.features[0] = nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3, bias=Fal
```

In [ ]:
```
modelo = chexnet.to(device)
```

Entrenamiento y validación del modelo

In [ ]:
```
num_epochs = 10
for epoch in range(num_epochs):
  model.train()
  total_loss = 0.0
  for data, targets in train_loader:
    data = data.to(device)
    targets = targets.to(device)
    optimizador.zero_grad()
    out = model(data)
    loss = criterio(out, targets.unsqueeze(1))
    loss.backward()
    optimizador.step()
    total_loss += loss.item()

    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {total_loss/len(train_loader)}")

print('Training finished')
```

In [ ]:
```
model.eval()
total_correct = 0
total_samples = 0
```

```
with torch.no_grad():
    for data, targets in test_loader:
        data = data.to(device)
        targets = targets.to(device)
        outputs = model(data)
        targets = targets.unsqueeze(1).float()
        predictions = (outputs > 0.5).float()
        correct = (predictions == targets).sum().item()
        total_correct += correct
        total_samples += targets.size(0)

# Calcular la precisión en el conjunto de prueba
accuracy = (total_correct / total_samples) * 100.0
print(f'Accuracy on test set: {accuracy:.2f}%')
```

```
Accuracy on test set: 64.88%
```

Inferencia

El siguiente codigo prueba el modelo entrenado y guardado

In [ ]:

```
modelo = CNN().to(device)
modelo.load_state_dict(torch.load('modelo.pth'))
modelo.eval()

def predict_image(image_path):
    image = Image.open(image_path)
    image_tensor = transform(image).unsqueeze(0)
    image_tensor = image_tensor.to(device)

    with torch.no_grad():
        output = modelo(image_tensor)
        predicted_class = torch.argmax(output).item()

    return predicted_class

image_path = 'images_0/19.png'
predicted_class = predict_image(image_path)
result = "No Trauma" if predicted_class == 0 else "Trauma"
plt.imshow(Image.open(image_path))
plt.axis("off")
print('Prediction:', result)
```

```
Prediction: No Trauma
```