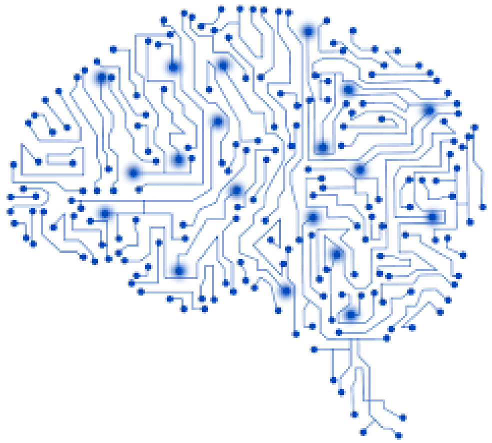
University of Minho
School of Engineering

ISLab



Aprendizagem Profunda

Modelos generativos

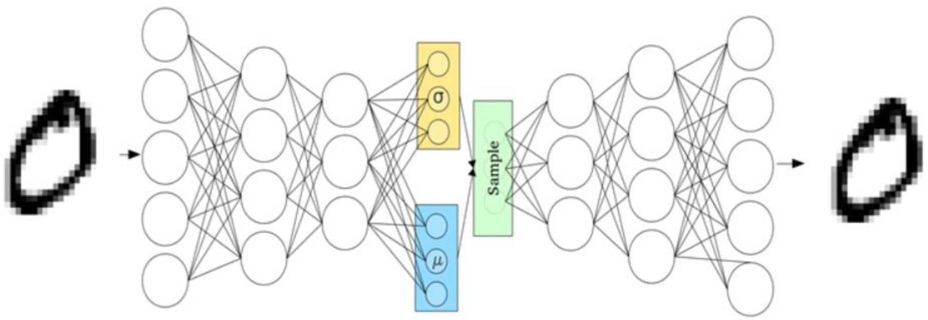
APP @ MEI/1º ano – 2º Semestre
Victor Alves

Part VIII

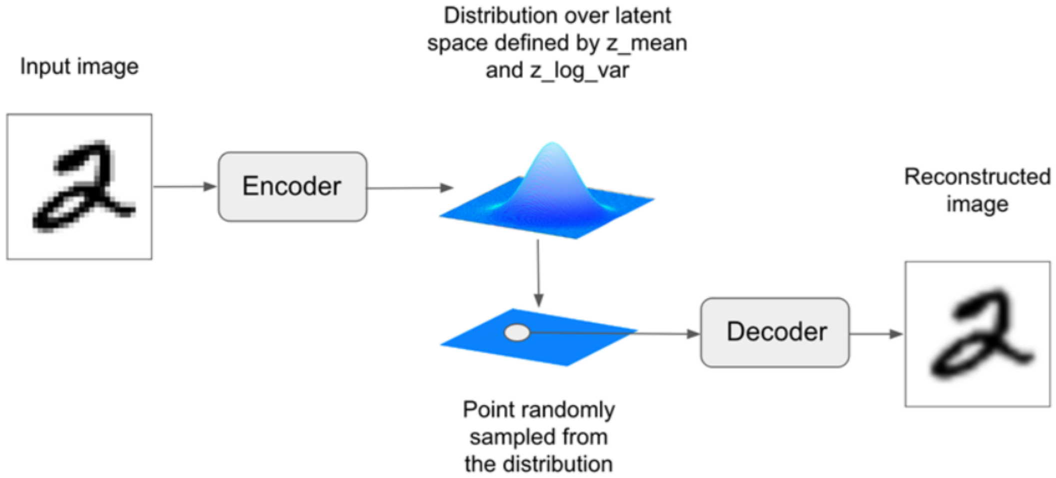
Contents

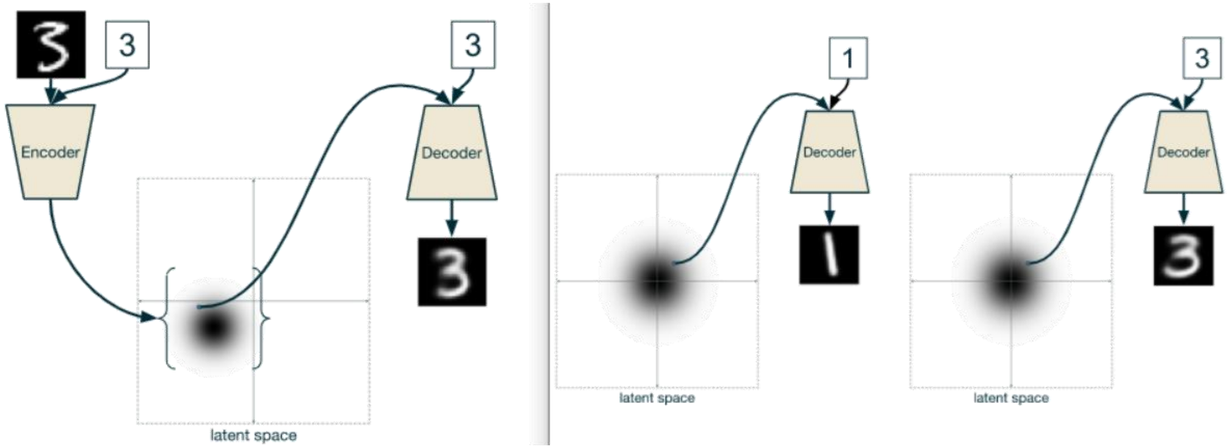
- CVAE
- GANs
- FCN

Hands On



- Conditional Variational Autoencoder with MNIST dataset
  - 11_pyt_CVAE_MLP_treino_MNIST.ipynb
  - 12_pyt_CVAE_MLP_generate_MNIST.ipynb





Hands On

11_pyt_CVAE_MLP_treino_MNIST.ipynb

2. Definir o modelo

```
import models_mnist #modulo python com os modelos


# definir a rede neuronal
model = models_mnist.CVAE_MLP(x_dim=784, h_dim1= 512, h_dim2=256, z_dim=2, c_dim=10)
#visualizar a rede
print(summary(model, [(BATCH_SIZE, 784), (BATCH_SIZE,10)], dtypes=[torch.float, torch.long], verbose=0))
#summary(model, [(1, 300), (1, 300)], dtypes=[torch.float, torch.long])
model.to(device)
```

Hands On

11_pyt_CVAE_MLP_treino_MNIST.ipynb

3. Treinar o modelo

```
def train_model(h5_file,train_dl, test_dl, model, loss_function,
optimizer, scheduler, epochs):
    liveloss = PlotLosses()
    for epoch in range(epochs):
        logs = {}
        model.train()
        running_loss  = 0.0
        for inputs, cond in train_dl:
            inputs = inputs.to(device)
            cond = one_hot(cond, 10).to(device)
            outputs, mu, log_var, _ = model(inputs, cond)
            loss = loss_function(outputs, inputs, mu, log_var)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        epoch_loss = running_loss / len(train_dl.dataset)
        logs['loss'] = epoch_loss*1000
        model.eval()
        running_loss  = 0.0
        ...
```

```
        ...
        for inputs, cond in test_dl:
            inputs = inputs.to(device)
            cond = one_hot(cond, 10).to(device)
            outputs, mu, log_var, _ = model(inputs, cond)
            loss = loss_function(outputs, inputs, mu, log_var)
            running_loss += loss.item()
        epoch_loss = running_loss / len(test_dl.dataset)
        logs['val_loss'] = epoch_loss*1000
        scheduler.step(epoch_loss) #callback a meio para atualizar lr
        epoch_lr = optimizer.param_groups[0]['lr']
        logs['val_lr'] = epoch_lr
        liveloss.update(logs)
        liveloss.send()
    torch.save(model,h5_file)
```

Hands On

11_pyt_CVAE_MLP_treino_MNIST.ipynb

```
EPOCHS = 50
LEARNING_RATE = 1e-3

def loss_function(recon_x, x, mu, log_var):
    BCE = F.binary_cross_entropy(recon_x, x.view(-1, 784), reduction='sum')
    KLD = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())
    return BCE + KLD

from torch.autograd import Variable
def one_hot(labels, class_size):  #one-hot encoding
    targets = torch.zeros(labels.size(0), class_size)
    for i, label in enumerate(labels):
        targets[i, label] = 1
    return targets

optimizer = Adam(model.parameters(), lr=LEARNING_RATE)
scheduler = ReduceLROnPlateau(optimizer, 'min', factor=0.1, patience=5)
starttime = time.perf_counter()
train_model('CVAE_MLP_MNIST.pth', train_dl, test_dl, model, loss_function, optimizer, scheduler, EPOCHS)
endtime = time.perf_counter()
print(f"Tempo gasto: {endtime - starttime} segundos")
```

Hands On

11_pyt_CVAE_MLP_treino_MNIST.ipynb

4. Usar o Autoencoder

```
def visualize(input_imgs, output_imgs):
    input_imgs=input_imgs.permute((1, 2, 0))
    output_imgs=output_imgs.permute((1, 2, 0))
    plt.subplots(1,2, figsize=(10, 10))
    plt.subplot(1,2,1)
    plt.axis('off')
    plt.grid(b=None)
    plt.title('Autoencoder Input')
    plt.imshow(input_imgs, cmap='gray')
    plt.subplot(1,2,2)
    plt.axis('off')
    plt.grid(b=None)
    plt.title('Autoencoder Output')
    plt.imshow(output_imgs, cmap='gray')
    plt.show()


model= torch.load('CVAE_MLP_MNIST.pth')
inputs, outputs = test_image_reconstruction(model, train_dl)
visualize(inputs, outputs)
```

```
def test_image_reconstruction(model, test_dl):
    for img,cond in test_dl:
        img = img.to(device)
        cond = one_hot(cond, 10).to(device)
        img = img.view(img.size(0), -1)
        print(img.shape)
        outputs,_,_,_ = model(img,cond)
        print(outputs.shape)
        outputs = outputs.view(outputs.size(0), 1, 28, 28).cpu().data
        print(outputs.shape)
        inputs = img.view(outputs.size(0), 1, 28, 28).cpu().data
        save_image(outputs, 'mnist_reconstruction_out.png')
        save_image(inputs, 'mnist_reconstruction_in.png')
        outputs = make_grid(outputs)
        inputs = make_grid(inputs)
        break
    return inputs, outputs
```

Hands On

11_pyt_CVAE_MLP_treino_MNIST.ipynb

```
#fazer uma previsão utilizando um caso
def make_prediction(model, img_list, cond, idx): #img shape (784,1)
    print(img_list.shape)
    print(img_list.dtype)
    img_list = img_list.to(device)
    cond = one_hot(cond, 10).to(device)
    prediction, _, _,_ = model(img_list,cond)
    print(prediction.shape)
    prediction = prediction.view(prediction.size(0), 1, 28,
28).cpu().data
    print(prediction.shape)
    img = img_list[idx].reshape(1,28, 28).cpu() )
    plt.subplots(1,2, figsize=(10, 10))
    plt.subplot(1,2,1)
    plt.axis('off')
    plt.grid(b=None)
    plt.title('Imagem Input')
    plt.imshow(img.permute((1, 2, 0)), cmap='gray')
    plt.subplot(1,2,2)
    ...
```

```
    ...
    plt.axis('off')
    plt.grid(b=None)
    plt.title('Imagem Output')
    plt.imshow(prediction[idx].permute((1, 2, 0)), cmap='gray')
    plt.show()

_, (inputs, targets) = next(enumerate(test_dl))
make_prediction(model,inputs,targets, 10)
```

Hands On

12_pyt_CVAE_MLP_generate_MNIST.ipynb

2. Definir o modelo

```
import models_mnist #modulo python com os modelos

# definir a rede neuronal
model = models_mnist.CVAE_MLP(x_dim=784, h_dim1= 512, h_dim2=256, z_dim=2, c_dim=10)
#visualizar a rede
print(summary(model, [(BATCH_SIZE, 784), (BATCH_SIZE,10)], dtypes=[torch.float, torch.long], verbose=0))
#summary(model, [(1, 300), (1, 300)], dtypes=[torch.float, torch.long])
model.to(device)



from torch.autograd import Variable
def one_hot(labels, class_size):  #one-hot encoding
    targets = torch.zeros(labels.size(0), class_size)
    for i, label in enumerate(labels):
        targets[i, label] = 1
    return targets #Variable(targets)
```

Hands On

12_pyt_CVAE_MLP_generate_MNIST.ipynb

3. Usar o Autoencoder

```
def visualize(input_imgs, output_imgs):
    input_imgs=input_imgs.permute((1, 2, 0))
    output_imgs=output_imgs.permute((1, 2, 0))
    plt.subplots(1,2, figsize=(10, 10))
    plt.subplot(1,2,1)
    plt.axis('off')
    plt.grid(b=None)
    plt.title('Autoencoder Input')
    plt.imshow(input_imgs, cmap='gray')
    plt.subplot(1,2,2)
    plt.axis('off')
    plt.grid(b=None)
    plt.title('Autoencoder Output')
    plt.imshow(output_imgs, cmap='gray')
    plt.show()
```

```
def test_image_reconstruction(model, test_dl):
    for img,cond in test_dl:
        img = img.to(device)
        cond = one_hot(cond, 10).to(device)
        img = img.view(img.size(0), -1)
        print(img.shape)
        outputs,_,_,_ = model(img,cond)
        print(outputs.shape)
        outputs = outputs.view(outputs.size(0), 1, 28, 28).cpu().data
        print(outputs.shape)
        inputs = img.view(outputs.size(0), 1, 28, 28).cpu().data
        save_image(outputs, 'mnist_reconstruction_out.png')
        save_image(inputs, 'mnist_reconstruction_in.png')
        outputs = make_grid(outputs)
        inputs = make_grid(inputs)
        break
    return inputs, outputs
```

```
model= torch.load('CVAE_MLP_MNIST.pth')
inputs, outputs = test_image_reconstruction(model, train_dl)
visualize(inputs, outputs)
```

Hands On

12_pyt_CVAE_MLP_generate_MNIST.ipynb

4. Avaliar o modelo

```
#fazer uma previsão utilizando um caso
def make_prediction(model, img_list, cond, idx): #img shape (784,1)
    print(img_list.shape)
    print(img_list.dtype)
    img_list = img_list.to(device)
    cond = one_hot(cond, 10).to(device)
    prediction, _, _,_ = model(img_list,cond)
    print(prediction.shape)
    prediction = prediction.view(prediction.size(0), 1, 28,
28).cpu().data
    print(prediction.shape)
    img = img_list[idx].reshape(1,28, 28).cpu() )
    plt.subplots(1,2, figsize=(10, 10))
    plt.subplot(1,2,1)
    plt.axis('off')
    plt.grid(b=None)
    plt.title('Imagem Input')
    plt.imshow(img.permute((1, 2, 0)), cmap='gray')
    plt.subplot(1,2,2)
    ...
```

```
    ...
    plt.axis('off')
    plt.grid(b=None)
    plt.title('Imagem Output')
    plt.imshow(prediction[idx].permute((1, 2, 0)), cmap='gray')
    plt.show()

_, (inputs, targets) = next(enumerate(test_dl))
make_prediction(model,inputs,targets, 10)
```

Hands On

12_pyt_CVAE_MLP_generate_MNIST.ipynb

5. Gerar uma imagem

```
import torch
def generate_digit(model, digit):
    print(digit)
    tensor_zeros = torch.zeros(1, 10)
    tensor_zeros[0,digit] = 1 #encoding do digit
    digit = tensor_zeros.to(device)
    print(digit)
    z = torch.randn(1, 2).to(device) #.cuda()#gerar 1 latent space 
aleatorio
    prediction1 = model.decoder(z,digit)
    prediction1 = prediction1.view(prediction1.size(0), 1, 28, 
28).cpu().data
    z = torch.randn(1, 2).to(device) #.cuda()
    prediction2 = model.decoder(z,digit)
    prediction2 = prediction2.view(prediction2.size(0), 1, 28, 
28).cpu().data
    plt.subplots(1,2, figsize=(10, 10))
    plt.subplot(1,2,1)
    plt.axis('off')
    plt.grid(b=None)
    plt.title('Imagem gerada 1')
    ...
```

```
    ...
    plt.imshow(prediction1[0].permute((1, 2, 0)), cmap='gray')
    plt.subplot(1,2,2)
    plt.axis('off')
    plt.grid(b=None)
    plt.title('Imagem gerada 2')
    plt.imshow(prediction2[0].permute((1, 2, 0)), cmap='gray')
    plt.show()

def generate_all_digits(model):
    z = torch.randn(10, 2).to(device) #.cuda()
    c = torch.eye(10).to(device) #.cuda()#gerar labels dos 10 digitos
    sample = model.decoder(z, c)
    print(sample.shape)
    sample = sample.view(sample.size(0), 1, 28, 28).cpu().data
    print(sample.shape)
    sample = make_grid(sample, nrow=5)
    print(sample.shape)
    plt.figure(figsize=(15, 15))
    plt.imshow(sample.permute((1, 2, 0)))
```

```
generate_digit(model, 0) #receives an image tensor with shape (784,1)
```

Hands On

- Generative Adversarial Networks with medMNIST dataset
  - 13_pyt_GAN_medNIST.ipynb

Using MONAI to train a network to generate images from a random input tensor. A simple GAN is used to act with separate Generator and Discriminator networks.This will go through the steps of:

- Loading data from a remote source
- Building a dataset from this data and transforming it
- Defining the networks
- Training and evaluation

Hands On

13_pyt_GAN_medNIST

Setting deterministic training for reproducibility

Turning this off will start the random state of the notebook at some arbitrary state; setting the seed will give reproducibility for random transformations between runs.

```
set_determinism(seed=0)
```

Definition of training variables

```
disc_train_interval = 1
disc_train_steps = 5
batch_size = 300
latent_size = 64
num_epochs = 50
real_label = 1
gen_label = 0
learning_rate = 2e-4
betas = (0.5, 0.999)
```

1.1. Get the dataset

Download the dataset

The MedNIST dataset was collected from several TCIA datasets, thr RSNA Bone Age Challenge and the NIH chest X-ray dataset.

The dataset was made available by Dr. Bradley J. Erickson M.D., Ph.D. (Department of Radiology, Mayo Clinic) under Creative Commons CC BY-SA 4.0 license.

Download links:

- https://www.dropbox.com/s/5wwskxctvcxiuea/MedNIST.tar.gz
- https://reposlink.di.uminho.pt/uploads/8920e974b5709bd3deafe02d18076229.file.MedNIST.tar.gz

Hands On

13_pyt_GAN_medNIST

To load the actual image data from the tar file, we define a transformation type using Matplotlib. This is used with other transforms to prepare the data, followed by random augmentation transforms. The `CacheDataset` class is used here to store all the prepared images from the tarball, so we will have in memory all the prepared images ready to be augmented with random rotation, rotation and zoom operations:

```
class LoadTarJpeg(Transform):
    def __call__(self, data):
        if MIVBOX: #neste caso não precisa de fazer o extract do tar
            return plt.imread(data)
        else:
            return plt.imread(tar.extractfile(data))

train_transforms = Compose( [
        LoadTarJpeg(),
        AddChannel(),
        ScaleIntensity(),
        RandRotate(range_x=15, prob=0.5, keep_size=True),
        RandFlip(spatial_axis=0, prob=0.5),
        RandZoom(min_zoom=0.9, max_zoom=1.1, prob=0.5),
        ToTensor(),
    ]
)
```

```
train_ds = CacheDataset(hands, train_transforms)
train_loader = torch.utils.data.DataLoader(
    train_ds, batch_size=batch_size, shuffle=True, num_workers=0
)
```

Hands On

13_pyt_GAN_medNIST

2.1 Defining the Generator and Discriminator

We define our generator and discriminator networks. The parameters are carefully chosen to suit the image size of `(1, 64, 64)` as loaded from the tar file. The input images to the discriminator are reduced four times to produce very small images, which are flattened and passed as input to a fully connected layer. The input latent vectors to the generator are passed through a fully connected layer to produce an output of the form `(64, 8, 8)`. This is then subsampled three times to produce a final output that has the same shape as the real images. The networks are initialised with a normalisation scheme to improve the results:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
disc_net = Discriminator(
    in_shape=(1, 64, 64),
    channels=(8, 16, 32, 64, 1),
    strides=(2, 2, 2, 2, 1),
    num_res_units=1,
    kernel_size=5,
).to(device)
gen_net = Generator(
    latent_shape=latent_size, start_shape=(64, 8, 8), channels=[32, 16, 8, 1], strides=[2, 2, 2, 1],
)
disc_net.apply(normal_init)
gen_net.apply(normal_init)
gen_net.conv.add_module("activation", torch.nn.Sigmoid())
gen_net = gen_net.to(device)
```

Hands On

13_pyt_GAN_medNIST

2.2. Defining the loss functions and optimizers

We now define the loss functions to be used with helper functions to complete the loss calculation process for the generator and discriminator. We have also defined our optimisers:

```
disc_loss = torch.nn.BCELoss()
gen_loss = torch.nn.BCELoss()
disc_opt = torch.optim.Adam(disc_net.parameters(), learning_rate, betas=betas)
gen_opt = torch.optim.Adam(gen_net.parameters(), learning_rate, betas=betas)

def discriminator_loss(gen_images, real_images):
    #The discriminator loss if calculated by comparing its prediction for real and generated images.
    real = real_images.new_full((real_images.shape[0], 1), real_label)
    gen = gen_images.new_full((gen_images.shape[0], 1), gen_label)
    realloss = disc_loss(disc_net(real_images), real)
    genloss = disc_loss(disc_net(gen_images.detach()), gen)
    return (realloss + genloss) / 2

def generator_loss(input):
    #The generator loss is calculated by determining how well the discriminator was fooled by the generated images.
    output = disc_net(input)
    cats = output.new_full(output.shape, real_label)
    return gen_loss(output, cats)
```

Hands On

13_pyt_GAN_medNIST

3. Training the model

We now train by iteration over the dataset for several epochs. At certain points after the generator training phase for each batch, the discriminator is trained for a series of steps over the same real and generated images.

```
epoch_loss_values = [(0, 0)]
gen_step_loss = []
disc_step_loss = []
step = 0
for epoch in range(num_epochs):
    gen_net.train()
    disc_net.train()
    epoch_loss = 0
    for i, batch_data in enumerate(train_loader):
        progress_msg=f"epoch {epoch + 1}, avg loss: {epoch_loss_values[-1][1]:.4f}"
        progress_bar(i, len(train_loader), progress_msg)
        real_images = batch_data.to(device)
        latent = torch.randn(real_images.shape[0], latent_size).to(device)
        gen_opt.zero_grad()
        gen_images = gen_net(latent)
        loss = generator_loss(gen_images)
        loss.backward()
        gen_opt.step()
```

```
        ...
        epoch_loss += loss.item()
        gen_step_loss.append((step, loss.item()))
        if step % disc_train_interval == 0:
            disc_total_loss = 0
            for _ in range(disc_train_steps):
                disc_opt.zero_grad()
                dloss = discriminator_loss(gen_images, real_images)
                dloss.backward()
                disc_opt.step()
                disc_total_loss += dloss.item()
            disc_step_loss.append((step, disc_total_loss / disc_train_steps))
        step += 1
    epoch_loss /= step
    epoch_loss_values.append((step, epoch_loss))
    IPython.display.clear_output()
```

Hands On

13_pyt_GAN_medNIST

The separate loss values for the generator and the discriminator can be viewed together. These should reach an equilibrium, as the ability of the generator to fool the discriminator balances with the ability of the network to accurately discriminate between real and false images.

```
plt.figure(figsize=(12, 5))
plt.semilogy(*zip(*gen_step_loss), label="Generator Loss")
plt.semilogy(*zip(*disc_step_loss), label="Discriminator Loss")
plt.grid(True, "both", "both")
plt.legend()
```

Hands On

13_pyt_GAN_medNIST

4. Sampling

Finally, we show some randomly generated images. Hopefully, most images have four fingers and a thumb, as expected (assuming that polydactyl examples were not present in large numbers in the dataset). This demo notebook does not train the networks for a long time - training beyond the predefined 50 epochs should improve the results.

```
test_size = 10
test_latent = torch.randn(test_size, latent_size).to(device)

test_images = gen_net(test_latent)

fig, axs = plt.subplots(1, test_size, figsize=(20, 4))

for i, ax in enumerate(axs):
    ax.axis("off")
    ax.imshow(test_images[i, 0].cpu().data.numpy(), cmap="gray")
```

Hands On

- Image classification with the MedNIST dataset
  - 14_pyt_FCN_medNIST.ipynb

Example of end-to-end training and evaluation based on the MedNIST dataset.

We will go through the following steps:

- Create a MONAI dataset for training and testing.
- Use MONAI transformations to preprocess data
- Use MONAI's DenseNet for the classification task
- Train the model with a PyTorch program
- Evaluate on the test dataset

Adapted from https://github.com/Project-MONAI/tutorials/blob/master/2d_classification/mednist_tutorial.ipynb

Hands On

14_pyt_FCN_medNIST

1.1. Get the dataset

Download the dataset

The MedNIST dataset was collected from several TCIA datasets, thr RSNA Bone Age Challenge and the NIH chest X-ray dataset.

The dataset was made available by Dr. Bradley J. Erickson M.D., Ph.D. (Department of Radiology, Mayo Clinic) under Creative Commons CC BY-SA 4.0 license.

Download links:

- https://www.dropbox.com/s/5wwskxctvcxiuea/MedNIST.tar.gz
- https://reposlink.di.uminho.pt/uploads/8920e974b5709bd3deafe02d18076229.file.MedNIST.tar.gz

Hands On

14_pyt_FCN_medNIST

1.3. Read image filenames from the dataset folders

First of all, check the dataset files and show some statistics.

There are 6 folders in the dataset: Hand, AbdomenCT, CXR, ChestCT, BreastMRI, HeadCT, which should be used as labels to train our classification model.

```
#conta os ficheiros do dataset por label
def file_list(dir_path):
    class_names = sorted([x for x in os.listdir(dir_path) if os.path.isdir(os.path.join(dir_path, x))])
    num_class = len(class_names)
    image_files = [[os.path.join(dir_path, class_name, x)
                    for x in os.listdir(os.path.join(dir_path, class_name))]
                   for class_name in class_names] #constrói uma lista de listas de ficheiros por diretoria de classe
    image_file_list = list()
    image_label_list = list()
    for i, class_name in enumerate(class_names): #para juntar as listas e construir a lista com os labels em numérico
        image_file_list.extend(image_files[i])
        image_label_list.extend([i] * len(image_files[i]))
    return image_file_list, image_label_list, class_names
...
```

```
...
image_file_list, image_label_list, class_names = file_list(DATA_DIR)
print('Total image count:', len(image_label_list))
image_width, image_height = Image.open(image_file_list[0]).size
print("Image dimensions:", image_width, "x", image_height)
labels_cout= [image_label_list.count(x) for x in set(image_label_list)]
for i,label in enumerate(class_names):
    print(f"Label: {label:11}  {labels_cout[i]:5d} cases")
```

Hands On

14_pyt_FCN_medNIST

1.4 Viewing some randomly chosen examples from the data set

```
def visualize_sample_images(image_file_list,image_label_list,class_names):
    plt.subplots(3, 3, figsize=(8, 8))
    for i,k in enumerate(np.random.randint(len(image_label_list), size=9)):
        im = Image.open(image_file_list[k])
        arr = np.array(im)
        plt.subplot(3, 3, i + 1)
        plt.axis('off')
        plt.grid(b=None)
        plt.xlabel(class_names[image_label_list[k]])
        plt.imshow(arr, cmap='gray', vmin=0, vmax=255)
    plt.tight_layout()
    plt.show()

visualize_sample_images(image_file_list,image_label_list,class_names)
```

Hands On

14_pyt_FCN_medNIST

1.5 Prepare training, validation and test data lists

Randomly select 10% of the data set as validation and 10% as test

```
#separar a lista de ficheiros em 3 partes para treino, validação e teste
def holdout_dataset(image_file_list,image_label_list):
    valid_frac, test_frac = 0.1, 0.1
    trainX, trainY = list(), list()
    valX, valY = list(), list()
    testX, testY = list(), list()
    for i in range(len(image_label_list)):
        rann = np.random.random()
        if rann < valid_frac:
            valX.append(image_file_list[i])
            valY.append(image_label_list[i])
        elif rann < test_frac + valid_frac:
            testX.append(image_file_list[i])
            testY.append(image_label_list[i])
        else:
            trainX.append(image_file_list[i])
            trainY.append(image_label_list[i])
    return trainX, trainY, valX, valY, testX, testY
...
```

```
...
trainX, trainY, valX, valY, testX, testY =
holdout_dataset(image_file_list,image_label_list)

print("Training count =",len(trainX))
print("Validation count =", len(valX))
print("Test count =",len(testX))
```

Hands On

14_pyt_FCN_medNIST

```
def visualize_holdout_balance(labels, class_names, titulo):
    sns.set_style('whitegrid')
    print("casos:",len(labels))
    x, y = np.unique(labels, return_counts=True)
    x_ext=[class_names[n]  for n in x]
    print(x_ext)
    print([str(n) for n in x])
    print(y)
    print(np.sum(y))
    grafico=sns.barplot(x_ext, y)
    grafico.set_title(f'Data balance: {titulo}')
    plt.xticks(rotation=70)
    plt.tight_layout()
    plt.show()

print("-----------------------------------casos_treino-----------------------------------")
visualize_holdout_balance(trainY, class_names, 'Treino')
print("-----------------------------------casos_validação-----------------------------------")
visualize_holdout_balance(valY, class_names, 'Validação')
print("-----------------------------------casos_teste-----------------------------------")
visualize_holdout_balance(testY, class_names, 'Teste')
```

Hands On

14_pyt_FCN_medNIST

2.1. Define MONAI, Dataset and Dataloader transformations to preprocess data

```
train_transforms = Compose([
    LoadImage(image_only=True),
    AddChannel(),
    ScaleIntensity(),
    RandRotate(range_x=15, prob=0.5, keep_size=True),
    RandFlip(spatial_axis=0, prob=0.5),
    RandZoom(min_zoom=0.9, max_zoom=1.1, prob=0.5, keep_size=True),
    ToTensor()
])

val_transforms = Compose([
    LoadImage(image_only=True),
    AddChannel(),
    ScaleIntensity(),
    ToTensor()
])
```

```
class MedNISTDataset(torch.utils.data.Dataset):
    def __init__(self, image_files, labels, transforms):
        self.image_files = image_files
        self.labels = labels
        self.transforms = transforms
    def __len__(self):
        return len(self.image_files)
    def __getitem__(self, index):
        return self.transforms(self.image_files[index]),
self.labels[index]

train_ds = MedNISTDataset(trainX, trainY, train_transforms)
train_dl = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=True,
num_workers=2)

val_ds = MedNISTDataset(valX, valY, val_transforms)
val_dl = DataLoader(val_ds, batch_size=BATCH_SIZE, num_workers=2)

test_ds = MedNISTDataset(testX, testY, val_transforms)
test_dl = DataLoader(test_ds, batch_size=BATCH_SIZE, shuffle=True,
num_workers=2)
```

Hands On

14_pyt_FCN_medNIST

2.2. Define network and optimizer

1. Set the learning rate of how much the model is updated per batch

2. Set the total epoch number, as we have mixed and random transformations, so the training data for each epoch will be different

And since this is just an introductory tutorial, let's train 4 epochs.

If you train 10 epochs, the model can achieve 100% accuracy on the test dataset.

3. Use MONAI's DenseNet and switch to the GPU device. This DenseNet can support 2D and 3D classification tasks

4. Use the Adam optimizer

```
model = densenet121(
                    spatial_dims=2,
                    in_channels=1,
                    out_channels=len(class_names)
                   ).to(device)

device = torch.device("cuda")
img=Image.open(trainX[0])
print(np.array(img).shape) #apenas um canal
print(summary(model, input_size=(BATCH_SIZE, 1,64,64), verbose=0))
```

Hands On

14_pyt_FCN_medNIST

3. Training the model

Run a typical PyTorch training that runs the epoch loop and the step loop, and perform validation after each epoch.

Save the model weights to the archive if you get the best validation accuracy.

```
def train_model(h5_file, train_dl, val_dl, model, loss_function, 
optimizer, epochs):
    liveloss = PlotLosses()
    best_metric = -1
    best_metric_epoch = -1
    #epoch_loss_values = list()
    metric_values = list()

    for epoch in range(epochs):
        logs = {}
        model.train()
        epoch_loss = 0
        running_loss  = 0.0
        running_corrects  = 0.0
        ...
```

```
        ...
        for inputs, labels in train_dl:
            inputs = inputs.to(device)
            labels = labels.to(device)
            outputs = model(inputs)
            loss = loss_function(outputs, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            running_loss += loss.detach() * inputs.size(0)
            _, preds = torch.max(outputs, 1) # Getpredictions from the 
maximum value
            running_corrects += torch.sum(preds == labels.data)
        epoch_loss = running_loss / len(train_dl.dataset)
        epoch_acc = running_corrects.float() / len(train_dl.dataset)
        logs['loss'] = epoch_loss.item()
        logs['accuracy'] = epoch_acc.item()
        ...
```

Hands On

14_pyt_FCN_medNIST

```
        ...
        model.eval()
        running_loss  = 0.0
        running_corrects  = 0.0
        with torch.no_grad():
            y_pred = torch.tensor([], dtype=torch.float32, device=device)
            y = torch.tensor([], dtype=torch.long, device=device)
            for val_images, val_labels in val_dl:
                val_images = val_images.to(device)
                val_labels = val_labels.to(device)
                outputs = model(val_images)
                loss = loss_function(outputs, val_labels)
                running_loss += loss.detach() * val_images.size(0)
                _, preds = torch.max(outputs, 1)
                running_corrects += torch.sum(preds == val_labels.data)
                y_pred = torch.cat([y_pred, outputs], dim=0)
                y = torch.cat([y, val_labels], dim=0)
            epoch_loss = running_loss / len(val_dl.dataset)
            epoch_acc = running_corrects.float() / len(val_dl.dataset)
            logs['val_loss'] = epoch_loss.item()
            logs['val_accuracy'] = epoch_acc.item()
```

```
            ...
            auc_metric = compute_roc_auc(y_pred, F.one_hot(y, num_classes=6), average="none")
            auc_metric_m = np.mean(auc_metric)
            metric_values.append(auc_metric_m)
            acc_value = torch.eq(y_pred.argmax(dim=1), y)
            acc_metric = acc_value.sum().item() / len(acc_value)
            if auc_metric_m > best_metric:
                best_metric = auc_metric_m
                best_metric_epoch = epoch + 1
                torch.save(model.state_dict(), 'best_metric_model.pth')
                print('saved new best metric model')
            print("current epoch:%d current AUC:%.4f current accuracy:%.4f best AUC:%.4f at epoch:%d"%(epoch+1,auc_metric_m,acc_metric,best_metric,best_metric_epoch))
        logs['val_AUC'] = auc_metric_m
        liveloss.update(logs)
        liveloss.send()
    print("train completed, best_metric:%.4f at epoch:%d"%(best_metric,best_metric_epoch))
```

Hands On

14_pyt_FCN_medNIST

```
#treino do modelo densenet121
EPOCHS = 4
LEARNING_RATE = 1e-5
loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), LEARNING_RATE)
epoch_num = 4
val_interval = 1

device = torch.device("cuda")
starttime = time.perf_counter()
train_model('densenet121.pth', train_dl, val_dl, model, loss_function, optimizer, EPOCHS)
endtime = time.perf_counter()
print(f"Tempo gasto: {endtime - starttime} segundos")
```

Hands On

14_pyt_FCN_medNIST

4. Evaluate the model on the test data set

After training and validation, we have obtained the best model in the validation test.

We need to evaluate the model on the test dataset to verify that it is robust and not overfitting.

We will use these predictions to generate a classification report.

Hands On

14_pyt_FCN_medNIST

Sensitivity to Occlusion

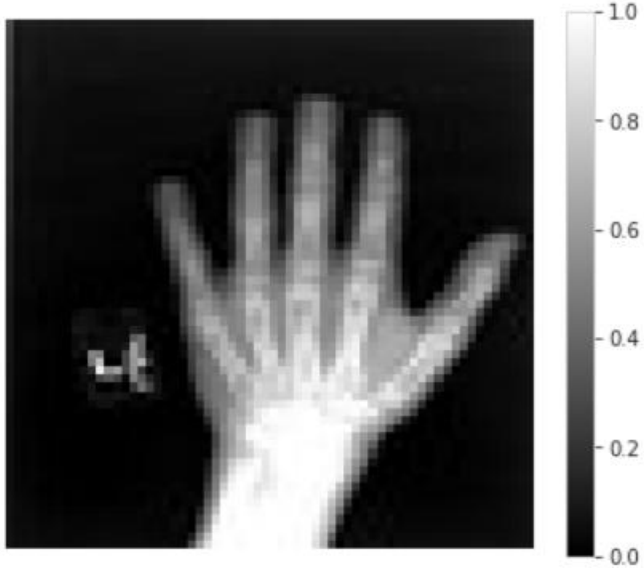
One method for trying to visualise why the net made a particular prediction is occlusion sensitivity. We occlude part of the image and see how the probability of a particular prediction changes. We then iterate over the image, moving the occluded part as we go, and in doing so build a sensitivity map detailing which areas were the most important in the decision making.

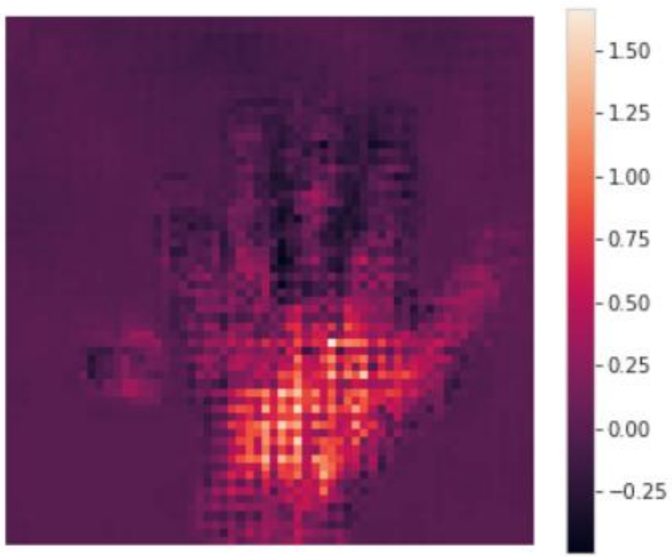
```
def get_rand_im():
    test_ds = MedNISTDataset(testX, testY, val_transforms)
    test_loader = torch.utils.data.DataLoader(test_ds, batch_size=1,
num_workers=2, shuffle=True)
    itera = iter(test_loader)
    test_data = next(itera)
    return test_data[0].to(device), test_data[1].unsqueeze(0).to(device)
```

```
def plot_occlusion_heatmap(im, heatmap):
    plt.subplots(1,2, figsize=(13, 5))
    plt.subplot(1,2,1)
    plt.axis('off')
    plt.grid(b=None)
    plt.imshow(np.squeeze(im.cpu()), cmap='gray')
    plt.colorbar()
    plt.subplot(1,2,2)
    plt.axis('off')
    plt.grid(b=None)
    plt.imshow(heatmap)
    plt.colorbar()
    plt.show()

device = torch.device("cuda")
#Get a random image and its corresponding label
im, label = get_rand_im()

#Get the occlusion sensitivity map
heatmap = occlusion_sensitivity.compute_occlusion_sensitivity(model, im,
label, margin=2)
plot_occlusion_heatmap(im, heatmap)
```

Hands On

14_pyt_FCN_medNIST





As cores claras correspondem a valores mais elevados, sendo uma evidencia para a classificação correta da imagem — Quando existe oclusão das áreas mais claras a metrica de avaliação para a classe correta diminui. Neste caso o centro da mão fornece a maior evidencia na classificação da figura como pertencendo à classe mão.

As área mais escuras correspondem a valores mais baixos, sendo uma evidencia para a classificação incorreta da imagem — Quando existe oclusão das áreas mais escuras a metrica de avaliação para a classe correta aumenta. Frequentemente estas areas são evidencias de uma outra classe e podem confundir o modelo

Hands On

Hands On

- Image classification with the MedNIST dataset
  - 15_pyt_FCN_medNIST.ipynb

Differences from the previous notebook:

- Use of CPU
- Evaluation step

Adapted from https://github.com/Project-MONAI/MONAI/blob/master/examples/notebooks/mednist_tutorial.ipynb.

Hands On

15_pyt_FCN_medNIST

2.2. Define network and optimizer

```
model = densenet121(
                    spatial_dims=2,
                    in_channels=1,
                    out_channels=len(class_names)
                   ).to(device)


img=Image.open(trainX[0])
print(np.array(img).shape) #apenas um canal
print(summary(model, input_size=(BATCH_SIZE, 1,64,64), verbose=0))
```

Hands On

15_pyt_FCN_medNIST

4. Evaluate the model on the test data set

After training and validation, we have obtained the best model in the validation test.

We need to evaluate the model on the test dataset to verify that it is robust and not overfitting.

We will use these predictions to generate a classification report.

```
def evaluate_model(test_dl, model):
    predictions = list()
    actual_values = list()
    for inputs, labels in test_dl:
        inputs = inputs.to(device)
        labels = labels.to(device)
        yprev = model(inputs)
        yprev = yprev.detach().cpu().numpy()
        actual = labels.cpu().numpy()
        yprev = np.argmax(yprev, axis=1)
        actual = actual.reshape((len(actual), 1))
        yprev = yprev.reshape((len(yprev), 1))
        predictions.append(yprev)
        actual_values.append(actual)
    predictions, actual_values = np.vstack(predictions),
np.vstack(actual_values)
    return actual_values, predictions
```

```
def display_predictions(actual_values, predictions ):
    acertou=0
    falhou = 0
    primeiros=0
    i=0
    for r,p in zip(actual_values, predictions):
        if primeiros <20:
            print(f'real:{r} previsão:{p}')
            primeiros +=1
        if r==p: acertou+=1
        else: falhou+=1
        i+=1
        if i>10:
            break
    ...
```

Hands On

15_pyt_FCN_medNIST

```
    ...
    corrects = np.sum(predictions == actual_values)
    acc = corrects / len(test_dl.dataset)
    acc = accuracy_score(actual_values, predictions)
    print(f'Accuracy: {acc:0.3f}\n')
    print(f'acertou:{acertou} falhou:{falhou}')
    acc = accuracy_score(actual_values, predictions)
    print(f'Accuracy: {acc:0.3f}\n')
    print(f'acertou:{acertou} falhou:{falhou}')

def display_confusion_matrix(cm,list_classes):
    plt.figure(figsize = (16,8))
    sns.heatmap(cm, annot=True, xticklabels=list_classes,
yticklabels=list_classes, annot_kws={"size": 12}, fmt='g',
linewidths=.5)
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()
```

```
model.load_state_dict(torch.load('best_metric_model.pth'))
model.eval()
actual_values, predictions = evaluate_model(test_dl, model)
print(len(actual_values))
display_predictions(actual_values, predictions)
print(classification_report(actual_values, predictions,
target_names=class_names, digits=4 ))
cr =classification_report(actual_values, predictions,
target_names=class_names, output_dict=True)
list_classes=[n for n in list(cr.keys())[0:6] ]
cm = confusion_matrix(actual_values, predictions)
print (cm)
display_confusion_matrix(cm,list_classes)
```

Hands On

- Segmentation Exercise
  - 16.1_pyt_exercicio_segmentacao.ipynb

In this exercise we will segment the left ventricle of the heart in relatively small images using neural nets.

The code for a segmentation net and its training is presented. The network is not very good, so t**he exercise is to improve the quality of segmentation by improving the network and/or the training scheme, including data loading efficiency and data augmentation**.

The data used here are derived from Sunnybrook Cardiac Dataset cardiac MR images, filtered to contain only segmentations of the left ventricular myocardium and reduced in XY dimensions.

Data extracted from:
https://github.com/ericspod/VPHSummerSchool2019/raw/master/scd_lvsegs.npz

Hands On

16.1_pyt_exercicio_segmentacao.ipynb

1. Data preparation

We now load the data from the remote source and view a sample:

```
data=np.load('scd_lvsegs.npz')
images = data["images"]
segs = data["segs"]
case_indices = data["caseIndices"]

images = images.astype(np.float32) / images.max()

print(images.shape, segs.shape, case_indices.shape)
plt.imshow(images[13] + (segs[13] * 0.25), cmap="gray")
plt.imshow(images[12] + segs[12], cmap="gray")
```

We split our data into a training and validation set, keeping the last 6 cases:

```
test_index = case_indices[-6, 0]  #keep the last 6 cases for testing

train_images, train_segs = images[:test_index], segs[:test_index]
test_images, test_segs = images[test_index:], segs[test_index:]
```

Hands On

16.1_pyt_exercicio_segmentacao.ipynb

We can now create a MONAI data loading object to compose batches during training and another for validation:

```
image_trans = Compose([ScaleIntensity(), AddChannel(), ToTensor()])
seg_trans = Compose([AddChannel(), ToTensor()])

ds = ArrayDataset(train_images, image_trans, train_segs, seg_trans)
loader = DataLoader(dataset=ds, batch_size=batch_size, num_workers=num_workers, pin_memory=pin_memory)

val_ds = ArrayDataset(test_images, image_trans, test_segs, seg_trans)
val_loader = DataLoader(dataset=val_ds, batch_size=batch_size, num_workers=num_workers, pin_memory=pin_memory)

im, seg = first(loader)
print(im.shape, im.min(), im.max(), seg.shape)
plt.imshow(im[0, 0].numpy() + seg[0, 0].numpy(), cmap="gray")
```

Hands On

16.1_pyt_exercicio_segmentacao.ipynb

2. Defining the model

We have now defined a simple network. This does not do a good job, so consider how to improve it by adding layers or other elements:

```
class SegNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            #layer 1: convolution, normalization, downsampling
            nn.Conv2d(1, 2, 3, 1, 1),
            nn.BatchNorm2d(2),
            nn.ReLU(),
            nn.MaxPool2d(3, 2, 1),
            #layer 2
            nn.Conv2d(2, 4, 3, 1, 1),
            #layer 3
            nn.ConvTranspose2d(4, 2, 3, 2, 1, 1),
            nn.BatchNorm2d(2),
            nn.ReLU(),
            #layer 4: output
            nn.Conv2d(2, 1, 3, 1, 1),
        )
```

```
def forward(self, x):
    return self.model(x)
```

Hands On

16.1_pyt_exercicio_segmentacao.ipynb

3. Model training

Training is very simple. For each epoch we train on each batch of images from the training set, thus training once with each image, and then evaluate with the validation set.

```
net = SegNet()
net = net.to(device)
opt = torch.optim.Adam(net.parameters(), lr)
loss = DiceLoss(sigmoid=True)
metric = DiceMetric(include_background=True, to_onehot_y=False,
sigmoid=True, reduction="mean" )
step_losses = []
epoch_metrics = []
total_step = 0
for epoch in range(num_epochs):
    net.train()
    for bimages, bsegs in loader:
        bimages = bimages.to(device)
        bsegs = bsegs.to(device)
        opt.zero_grad()
        prediction = net(bimages)
        loss_val = loss(prediction, bsegs)
        loss_val.backward()
        opt.step()
        ...
```

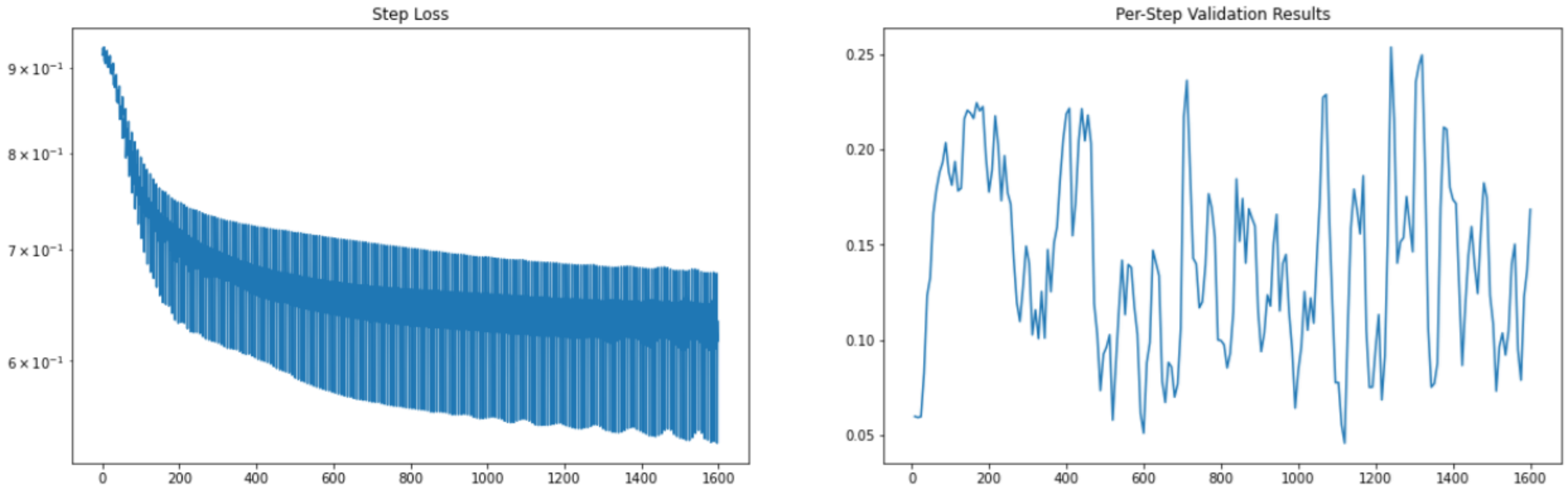
```
        ...
        step_losses.append((total_step, loss_val.item()))
        total_step += 1
    net.eval()
    metric_vals = []
    with torch.no_grad():
        for bimages, bsegs in val_loader:
            bimages = bimages.to(device)
            bsegs = bsegs.to(device)
            prediction = net(bimages)
            pred_metric = metric(prediction, bsegs)
            metric_vals.append(pred_metric.item())
    epoch_metrics.append((total_step, np.average(metric_vals)))
    progress_bar(epoch + 1, num_epochs, f"Validation Metric:
{epoch_metrics[-1][1]:.3}")
```

Hands On

16.1_pyt_exercicio_segmentacao.ipynb

Now, we graph the results of the training and find that the results are not very good:

```
fig, ax = plt.subplots(1, 2, figsize=(20, 6))

ax[0].semilogy(*zip(*step_losses))
ax[0].set_title("Step Loss")

ax[1].plot(*zip(*epoch_metrics))
ax[1].set_title("Per-Step Validation
Results")
plt.show()
```



As you can see, we are not getting good results with our network. The training loss values are jumping and they are not decreasing much anymore. The validation score reached 0.25, which is really poor.

→ The **next step** is to improve the results of our segmentation task. Things to consider changing include the network itself, how the data is loaded, how the plots can be composed and what transformations we want to use from MONAI.

Submit the improvement (optional)