Image Registration

we are going to perform registration using 2 different approachs. First we use classical SimpleITK. Then, we run a deep learning based approach which called VoxelMorph

Demands:

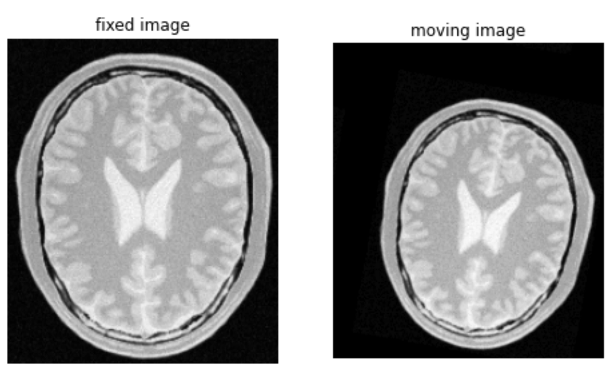
1. Define initial transform
2. Define Registration method
3. Set appropriate metric
4. Set optimizer
5. Set interpolator
6. Set resampler
7. print metric value during runnig
8. show final results

```
!pip install SimpleITK -q
```

```
     |████████████████████████████████| 48.4 MB 71.0 MB/s
```

```
import matplotlib.pyplot as plt
import SimpleITK as sitk
import cv2
import numpy as np
```

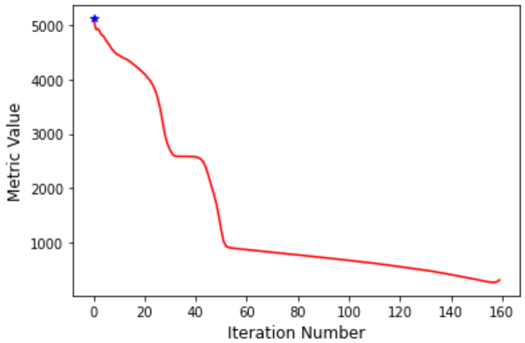
```
fixed_im =  sitk.ReadImage('./Q1_fixed.png', sitk.sitkFloat32)
moving_im =  sitk.ReadImage('./Q1_moving.png', sitk.sitkFloat32)
fixed_array = sitk.GetArrayViewFromImage(fixed_im)
moving_array = sitk.GetArrayViewFromImage(moving_im)

plt.subplots(1,2,figsize=(8,6))
plt.subplot(1,2,1)
plt.imshow(fixed_array, cmap='gray');
plt.title('fixed image')
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(moving_array,cmap='gray');
plt.title('moving image')
plt.axis('off')
plt.show()
```




```
def register_img(fixed_image, moving_image, use_affine = True, metric = 'MS', optimizer = 'GD'):

    # Define initial transform ---> Affine and Centered
    transform = sitk.AffineTransform(2) if use_affine else sitk.ScaleTransform(2)
    initial_transform = sitk.CenteredTransformInitializer(sitk.Cast(fixed_image,moving_image.GetPixelID()),
                                                          moving_image,
                                                          transform,
                                                          sitk.CenteredTransformInitializerFilter.GEOMETRY)

    # Define Registration method
```

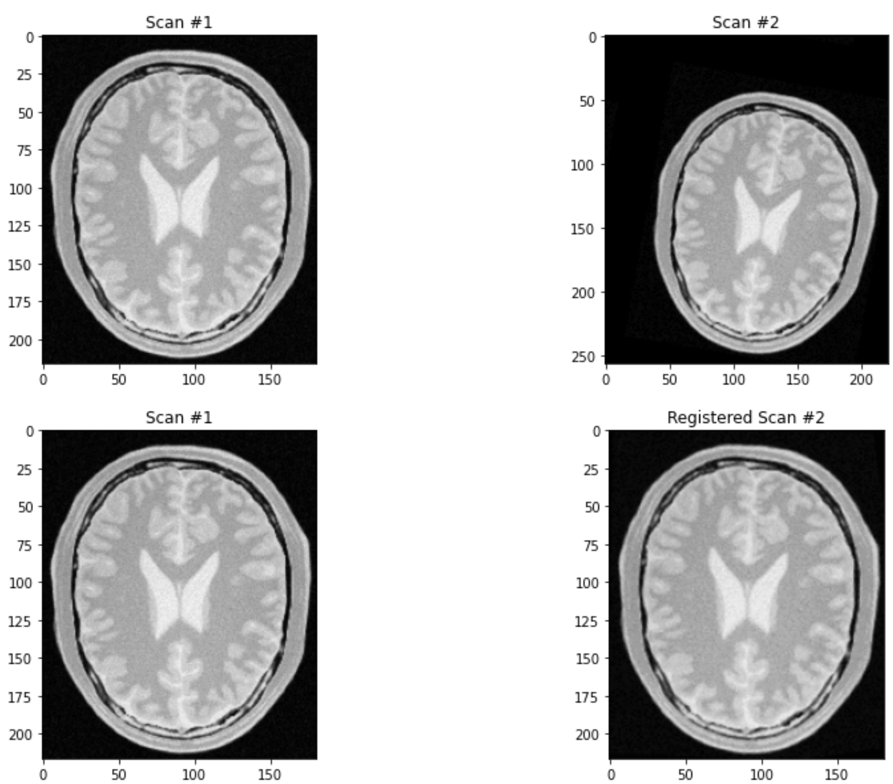
```
    registration_method = sitk.ImageRegistrationMethod()

    # Set appropriate metric --> Mean Square differences or MutualInformation
    if metric == 'MS':
        registration_method.SetMetricAsMeanSquares()
    else:
        registration_method.SetMetricAsMattesMutualInformation(numberOfHistogramBins=50)

    # Define Registration method --> Brute Force
    sample_per_axis = 12
    registration_method.SetOptimizerAsExhaustive([sample_per_axis//2,0,0])
    # Utilize the scale to set the step size for each dimension
    registration_method.SetOptimizerScales([2.0*3.14/sample_per_axis, 1.0,1.0])

    # Set interpolator
    registration_method.SetInterpolator(sitk.sitkLinear)

    # Set optimizer --> Gradient Descent or RegularStepGradientDescent
    if optimizer == 'GD':
        registration_method.SetOptimizerAsGradientDescent(learningRate=1.0,
                                                          numberOfIterations=200,
                                                          convergenceMinimumValue=1e-6,
                                                          convergenceWindowSize=10)
    else:
        registration_method.SetOptimizerAsRegularStepGradientDescent(learningRate=4.0,
                                                                     minStep=0.01,
                                                                     numberOfIterations=200,
                                                                     relaxationFactor=0.5)

    registration_method.SetOptimizerScalesFromPhysicalShift()
    registration_method.SetInitialTransform(initial_transform, inPlace=False)

    # print metric value during runnig --> Connect all of the observers so that we can perform plotting during registration
    registration_method.AddCommand(sitk.sitkStartEvent, start_plot)
    registration_method.AddCommand(sitk.sitkEndEvent, end_plot)
    registration_method.AddCommand(sitk.sitkMultiResolutionIterationEvent, update_multires_iterations)
    registration_method.AddCommand(sitk.sitkIterationEvent, lambda: plot_values(registration_method))

    ff_img = sitk.Cast(fixed_image, sitk.sitkFloat32)
    mv_img = sitk.Cast(moving_image, sitk.sitkFloat32)
    final_transform_v1 = registration_method.Execute(ff_img, mv_img)

    # show final results
    print('Optimizer\'s stopping condition, {0}'.format(registration_method.GetOptimizerStopConditionDescription()))
    print('Final metric value: {0}'.format(registration_method.GetMetricValue()))

    # Set resampler and interpolator
    resample = sitk.ResampleImageFilter()
    resample.SetReferenceImage(fixed_image)
    resample.SetInterpolator(sitk.sitkBSpline)
    resample.SetTransform(final_transform_v1)

    return sitk.GetArrayFromImage(resample.Execute(moving_image))


from IPython.display import clear_output
def start_plot():
    global metric_values, multires_iterations
    metric_values = []
    multires_iterations = []

def end_plot():
    global metric_values, multires_iterations
    del metric_values
    del multires_iterations
    plt.close()

# IterationEvent happens, update our data and display new figure
def plot_values(registration_method):
    global metric_values, multires_iterations

    metric_values.append(registration_method.GetMetricValue())
    clear_output(wait=True)
    # Plot the similarity metric values
    plt.plot(metric_values, 'r')
    plt.plot(multires_iterations, [metric_values[index] for index in multires_iterations], 'b*')
    plt.xlabel('Iteration Number',fontsize=12)
```

```
    plt.ylabel('Metric Value',fontsize=12)
    plt.show()


def update_multires_iterations():
    global metric_values, multires_iterations
    multires_iterations.append(len(metric_values))


registered = register_img(fixed_im, moving_im)
```




```
Optimizer's stopping condition, GradientDescentOptimizerv4Template: Convergence checker
Final metric value: 379.7428414157294
```

```
def plot_imgs(fixed_arr, moving_arr, registered_arr):
    fig, ((ax1, ax2), (ax1a, ax3)) = plt.subplots(2, 2, figsize = (14, 10))
    ax1.imshow(fixed_arr, cmap = 'gray', vmax = 255)
    ax1.set_title('Scan #1')
    ax2.imshow(moving_arr, cmap = 'gray', vmax = 255)
    ax2.set_title('Scan #2')
    ax1a.imshow(fixed_arr, cmap = 'gray', vmax = 255)
    ax1a.set_title('Scan #1')
    ax3.imshow(registered_arr, cmap = 'gray', vmax = 255)
    ax3.set_title('Registered Scan #2')


plot_imgs(fixed_array, moving_array, registered)
```



VoxelMorph

For dataset we are using FIRE dataset.

```
!pip install voxelmorph -q
```

```
     |████████████████████████████████| 75 kB 4.7 MB/s
     |████████████████████████████████| 86 kB 7.4 MB/s
```

```
import os, sys
import glob
import numpy as np
import random
import torch
import matplotlib.pyplot as plt
import·cv2
import·torch.nn.functional·as·F
import math
from torchvision import transforms
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader, Dataset, random_split
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
# List the image ids of each image
image_list=[".".join(f.split(".")[:-1]) for f in os.listdir('./drive/MyDrive/Q2_dataset') if os.path.isfile(os.path.join('./drive/MyDrive/Q2_
im_id = []
for i, item in enumerate(image_list):
    im_id.append(item.split('_')[0])
print(im_id)
```

```
['P10', 'P12', 'P11', 'P14', 'P13', 'P10', 'P11', 'P15', 'P13', 'P12', 'P15', 'P14', 'P16', 'P16', 'P19', 'P19', 'P20', 'P20', 'A10', 'A
```

```
os.environ['VXM_BACKEND'] = 'pytorch'
import voxelmorph as vxm

device = 'cuda'
os.environ['CUDA_VISIBLE_DEVICES'] = '0'
torch.backends.cudnn.deterministic = True
```

```
class Dataset(Dataset):

    def __init__(self, list_IDs):
        self.list_IDs = list_IDs

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

    def __getitem__(self, index):
        # Select sample
        ID = self.list_IDs[index]

        # Load data and get label
        fixed_image = torch.Tensor(cv2.resize(cv2.imread('./drive/MyDrive/Q2_dataset/' + ID + '_1.jpg', cv2.IMREAD_GRAYSCALE), (256, 256),
                                              interpolation=cv2.INTER_AREA))
        fixed_image = fixed_image = torch.unsqueeze(fixed_image, 2)
        moving_image = torch.Tensor(cv2.resize(cv2.imread('./drive/MyDrive/Q2_dataset/' + ID + '_2.jpg', cv2.IMREAD_GRAYSCALE), (256, 256),
                                               interpolation=cv2.INTER_AREA))
        moving_image = fixed_image = torch.unsqueeze(moving_image, 2)
        return moving_image, fixed_image
```

```
params = {'batch_size': 1, 'shuffle': True, 'num_workers': 2, 'worker_init_fn': np.random.seed(42)}
partition = {}
partition['train'], partition['validation'] = train_test_split(im_id, test_size = 0.33, random_state=42)

#Split dataset into Train and Test
training_set = Dataset(partition['train'])
training_generator = DataLoader(training_set, **params)
```

```
validation_set = Dataset(partition['validation'])
validation_generator = DataLoader(validation_set, **params)


# configure features
inshape = next(iter(training_generator))[0].shape[1:-1] # (256,256)
enc_nf = [16, 32, 32, 32]
dec_nf = [32, 32, 32, 32, 32, 16, 16]
# VoxelMorph network for (unsupervised) nonlinear registration between two images.
vm = vxm.networks.VxmDense(
    inshape=inshape,
    nb_unet_features=[enc_nf, dec_nf],
    bidir=False,
    int_downsize=1
    )
vm.to(device)
```

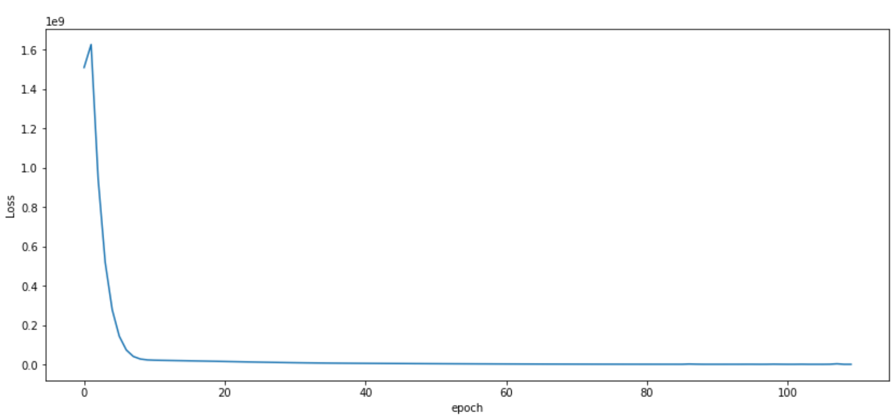
```
        (upsample): Upsample(scale_factor=2.0, mode=nearest)
        (downarm): ModuleList(
          (0): ConvBlock(
            (main): Conv2d(2, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
            (activation): LeakyReLU(negative_slope=0.2)
          )
          (1): ConvBlock(
            (main): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
            (activation): LeakyReLU(negative_slope=0.2)
          )
          (2): ConvBlock(
            (main): Conv2d(32, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
            (activation): LeakyReLU(negative_slope=0.2)
          )
          (3): ConvBlock(
            (main): Conv2d(32, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
            (activation): LeakyReLU(negative_slope=0.2)
          )
        )
        (uparm): ModuleList(
          (0): ConvBlock(
            (main): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (activation): LeakyReLU(negative_slope=0.2)
          )
          (1): ConvBlock(
            (main): Conv2d(64, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (activation): LeakyReLU(negative_slope=0.2)
          )
          (2): ConvBlock(
            (main): Conv2d(64, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (activation): LeakyReLU(negative_slope=0.2)
          )
          (3): ConvBlock(
            (main): Conv2d(48, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (activation): LeakyReLU(negative_slope=0.2)
          )
        )
        (extras): ModuleList(
          (0): ConvBlock(
            (main): Conv2d(34, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (activation): LeakyReLU(negative_slope=0.2)
          )
          (1): ConvBlock(
            (main): Conv2d(32, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (activation): LeakyReLU(negative_slope=0.2)
          )
          (2): ConvBlock(
            (main): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (activation): LeakyReLU(negative_slope=0.2)
          )
        )
      )
      (flow): Conv2d(16, 2, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (integrate): VecInt(
        (transformer): SpatialTransformer()
      )
      (transformer): SpatialTransformer()
    )
```

```
class Grad:

    def __init__(self, penalty='l1', loss_mult=None):
        self.penalty = penalty
        self.loss_mult = loss_mult
```

```
    def loss(self, _, y_pred):
        dy = torch.abs(y_pred[:, :, 1:, :] - y_pred[:, :, :-1, :])
        dx = torch.abs(y_pred[:, :, :, 1:] - y_pred[:, :, :, :-1])

        if self.penalty == 'l2':
            dy = dy * dy
            dx = dx * dx

        d = torch.mean(dx) + torch.mean(dy)
        grad = d / 3.0

        if self.loss_mult is not None:
            grad *= self.loss_mult
        return grad


optimizer = torch.optim.Adam(vm.parameters(), lr=1e-2)
image_loss_func = vxm.losses.NCC().loss
losses = [image_loss_func]
weights = [1]

# prepare deformation loss
losses += [Grad('l2', loss_mult=1).loss]
weights += [0.001]
each_epoch_loss = []
epochs = 110
# training loops
for epoch in range(0, epochs):
    epoch_total_loss = []
    epoch_loss = []
    for batch_moving, batch_fixed in training_generator:

        batch_moving = batch_moving.to(device).float().permute(0, 3, 1, 2)
        batch_fixed = batch_fixed.to(device).float().permute(0, 3, 1, 2)

        # run inputs through the model to produce a warped image and flow field
        y_pred = vm(batch_moving, batch_fixed)

        # calculate total loss
        loss = 0
        loss_list = []

        for n, loss_function in enumerate(losses):
            curr_loss = loss_function(batch_fixed, y_pred[n]) * weights[n]
            loss_list.append(curr_loss.item())
            loss += curr_loss

        epoch_loss.append(loss_list)
        epoch_total_loss.append(loss.item())

        # backpropagate and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    # print epoch info
    epoch_info = 'Epoch %d/%d' % (epoch + 1, epochs)
    losses_info = ', '.join(['%.4e' % f for f in np.mean(epoch_loss, axis=0)])
    loss_info = 'loss: %.4e  (%s)' % (np.mean(epoch_total_loss), losses_info)
    each_epoch_loss.append(np.mean(epoch_total_loss))
    print(' - '.join((epoch_info, loss_info)), flush=True)

# final model save
vm.save(os.path.join('./', '%04d.pt' % epochs))
```

```
Epoch 56/110 - loss: 2.3056e+06  (0.0000e+00, 2.3056e+06)
Epoch 57/110 - loss: 2.1646e+06  (0.0000e+00, 2.1646e+06)
Epoch 58/110 - loss: 2.0316e+06  (0.0000e+00, 2.0316e+06)
Epoch 59/110 - loss: 1.9012e+06  (0.0000e+00, 1.9012e+06)
Epoch 60/110 - loss: 1.7783e+06  (0.0000e+00, 1.7783e+06)
Epoch 61/110 - loss: 1.6610e+06  (0.0000e+00, 1.6610e+06)
Epoch 62/110 - loss: 1.5486e+06  (0.0000e+00, 1.5486e+06)
Epoch 63/110 - loss: 1.4419e+06  (0.0000e+00, 1.4419e+06)
Epoch 64/110 - loss: 1.3416e+06  (0.0000e+00, 1.3416e+06)
Epoch 65/110 - loss: 1.2467e+06  (0.0000e+00, 1.2467e+06)
Epoch 66/110 - loss: 1.1558e+06  (0.0000e+00, 1.1558e+06)
Epoch 67/110 - loss: 1.0717e+06  (0.0000e+00, 1.0717e+06)
Epoch 68/110 - loss: 9.9278e+05  (0.0000e+00, 9.9278e+05)
Epoch 69/110 - loss: 9.1831e+05  (0.0000e+00, 9.1831e+05)
Epoch 70/110 - loss: 8.5092e+05  (0.0000e+00, 8.5092e+05)
Epoch 71/110 - loss: 7.8621e+05  (0.0000e+00, 7.8621e+05)
Epoch 72/110 - loss: 7.2705e+05  (0.0000e+00, 7.2705e+05)
Epoch 73/110 - loss: 6.7222e+05  (0.0000e+00, 6.7222e+05)
Epoch 74/110 - loss: 6.2255e+05  (0.0000e+00, 6.2255e+05)
Epoch 75/110 - loss: 5.7555e+05  (0.0000e+00, 5.7555e+05)
Epoch 76/110 - loss: 5.3465e+05  (0.0000e+00, 5.3465e+05)
Epoch 77/110 - loss: 4.9657e+05  (0.0000e+00, 4.9657e+05)
Epoch 78/110 - loss: 4.5873e+05  (0.0000e+00, 4.5873e+05)
Epoch 79/110 - loss: 4.2468e+05  (0.0000e+00, 4.2468e+05)
Epoch 80/110 - loss: 3.9384e+05  (0.0000e+00, 3.9384e+05)
Epoch 81/110 - loss: 3.6989e+05  (0.0000e+00, 3.6989e+05)
Epoch 82/110 - loss: 3.4113e+05  (0.0000e+00, 3.4113e+05)
Epoch 83/110 - loss: 3.1679e+05  (0.0000e+00, 3.1679e+05)
Epoch 84/110 - loss: 3.0386e+05  (0.0000e+00, 3.0386e+05)
Epoch 85/110 - loss: 2.8408e+05  (0.0000e+00, 2.8408e+05)
Epoch 86/110 - loss: 2.5854e+05  (0.0000e+00, 2.5854e+05)
Epoch 87/110 - loss: 1.4625e+06  (0.0000e+00, 1.4625e+06)
Epoch 88/110 - loss: 6.6564e+05  (0.0000e+00, 6.6564e+05)
Epoch 89/110 - loss: 1.8865e+05  (0.0000e+00, 1.8865e+05)
Epoch 90/110 - loss: 1.7812e+05  (0.0000e+00, 1.7812e+05)
Epoch 91/110 - loss: 1.6372e+05  (0.0000e+00, 1.6372e+05)
Epoch 92/110 - loss: 1.8938e+05  (0.0000e+00, 1.8938e+05)
Epoch 93/110 - loss: 2.1993e+05  (0.0000e+00, 2.1993e+05)
Epoch 94/110 - loss: 2.1223e+05  (0.0000e+00, 2.1223e+05)
Epoch 95/110 - loss: 3.0771e+05  (0.0000e+00, 3.0771e+05)
Epoch 96/110 - loss: 3.3579e+05  (0.0000e+00, 3.3579e+05)
Epoch 97/110 - loss: 1.2476e+05  (0.0000e+00, 1.2476e+05)
Epoch 98/110 - loss: 1.7910e+05  (0.0000e+00, 1.7910e+05)
```

```
plt.figure(figsize = (14, 6))
plt.plot(each_epoch_loss)
plt.xlabel('epoch')
plt.ylabel('Loss')
plt.show()
```




```
model = vxm.networks.VxmDense.load('./0110.pt', device)
model.to(device)
model.eval()

epochtest_total_loss = []
epochtest_loss = []
for moving, fixed in validation_generator:
    # set up tensors and permute
    input_moving = moving.to(device).float().permute(0, 3, 1, 2)
```

```
        input_fixed = fixed.to(device).float().permute(0, 3, 1, 2)

        # predict
        pred = model(input_moving, input_fixed, registration=True)
        loss = 0
        loss_list = []

        for n, loss_function in enumerate(losses):
            curr_loss = loss_function(batch_fixed, y_pred[n]) * weights[n]
            loss_list.append(curr_loss.item())
            loss += curr_loss

        epochtest_loss.append(loss_list)
        epochtest_total_loss.append(loss.item())


    losses_info = ', '.join(['%.4e' % f for f in np.mean(epochtest_loss, axis=0)])
    loss_info = 'loss: %.4e  (%s)' % (np.mean(epochtest_total_loss), losses_info)
    print(loss_info, flush=True)
```

```
loss: 1.0056e+04  (0.0000e+00, 1.0056e+04)
```