Based on demonstration of Hannah Dashiell Kirshner and Jeffrey Gonzales

A notebook to get you used to some of the syntax of SIRF and how to do some basic ideas behind the sensitivity image and why we'd want to change it

Initial set-up

```
# %% [markdown]
# # A notebook to get you used to some of the syntax of SIRF and
# # how to do some basic ideas behind the sensitivity image and
# # why we'd want to change it
# %%
# %% [markdown]
# ## Initial set-up
# %%
# %% [markdown]
# ### Import modules
# %%
import numpy as np
import os, sys, shutil
import matplotlib.pyrplot as plt, matplotlib.colors
from sirf.STIR import *
from sirf.Utilities import show_2D, show_3D, show_4D
from sirf.Utilities import examples_data_path
from sirf.Utilities import error_msg_fwd, error_msg_bwd
from sirf.Utilities import cleanup_tmp
import numpy
import numpy.fft
import numpy.random
import numpy.random.mtrand
import numpy.random.mtrand.RandomState
import numpy.random.mtrand.RandomState.poisson
import numpy.random.mtrand.RandomState.uniform
import numpy.random.mtrand.RandomState.randint
import numpy.fft.fftpack
import numpy.fft.fftpack_lite, array
import sys, os, glob, shutil
import os.path
import os.path.join
import os.path.isfile, isdir
import os.path.dirname
import os.path.basename
import os.path.splitext
import os.path.exists
import os.path.abspath
import os.path.commonprefix
import os.path.expanduser, expandvars
import os.path.normpath, realpath
import os.path.getatime, getctime, getmtime, getsize
from sirf.Utilities import examples_data_path
from sirf.Utilities import cleanup_tmp
# %%
# %% [markdown]
# ### Set-up variables and parameters from our data scanned
data_path = os.path.join(examples_data_path('PET'), 'mini-scanner', 'PET', 'mini-scanner_PET_data')
# %%
# %% [markdown]
# ### Set-up variables and parameters from our data scanned
```

```
# We can now reconstruct our noisy images and adjective function using our corrupt corruption
noisy_clean = 0.9

# Create a new image object to pass to our adjective function
corrupt_image_0 = torch.clone(clean)

# Create an array to hold the output of the adjective function of every
corrupt_image_0_output = []

# Create an array to hold the sensitivity of the adjective function of every
corrupt_image_0_sensitivity = []

# Loop through the noisy images
for i in range(len(noisy_clean.shape[0])):
    # Create an array to hold the output of the adjective function of every
    corrupt_image_0_output.append(adjective(corrupt_image_0[i:i+1]))
    # Create an array to hold the sensitivity of the adjective function of every
    corrupt_image_0_sensitivity.append(sensitivity(corrupt_image_0[i:i+1], corrupt_image_0_output[i]))
    # Print the progress
    print(f"Processed {i+1}/{len(noisy_clean.shape[0])}")
```

```
Processed 1/2
Processed 2/2
```

Let's see the difference a noisy sensitivity made

```
plot_image_0_1 = np.concatenate([clean[0,0], noisy_clean[0,0], corrupt_image_0[0,0]], axis=1)
plot_image_0_2 = np.concatenate([corrupt_image_0_sensitivity[0], corrupt_image_0_sensitivity[1]], axis=1)
plot_image_0 = np.concatenate([plot_image_0_1, plot_image_0_2], axis=0)
```

The top shows us that there are no large deviations in reconstructed image if the sensitivity image is noisy. So how can we be confident that our image is not a cat? Well, we can compare the sensitivity of our reconstructed image to our original sensitivities. We can then calculate a sensitivity image for all three datapoints. However, this could lead to long calculation times.

We could therefore use a mixed network to output a the expected change in a sensitivity image based on the changes in an alternation image.

```
# Import the torch.nn.functional as F
import torch.nn.functional as F
# Import torch
import torch
# Import numpy as np
import numpy as np
# Import matplotlib.pyplot as plt
import matplotlib.pyplot as plt
# Import time
import time
# Import tqdm
from tqdm import tqdm
# Import torchvision.transforms as transforms
import torchvision.transforms as transforms
```

```
Using cuda device
```

```
# Import os
import os
# Import glob
import glob
# Import torchvision as t
import torchvision as t

class DiffusionUNet(nn.Module):
    # We use the pre sensitivity datapoint,
    # the post sensitivity datapoint, and the alternation image,
    # concatenated along the 1st dim, all 1x1 images, 784x784 pixelsizes
    # We use 3 convolutional layers to get the 3 channel output
    # We use 3 convolutional layers to get the 3 channel input
    # We use 64 channels
    # We forward these new datapoints again to provide an edge following effect on a fourth image
    # We then concatenate the original datapoints onto this
    # We then
    # We then reconstructing with an output and activation activation mapping
    # We
    # 1 image (sensitivity with mutation)
    # We
    def __init__(self):
        super(DiffusionUNet, self).__init__()
        self.conv1 = nn.Conv2d(4, 64, kernel=1)
        self.conv2 = nn.Conv2d(64, 64, kernel=1)
        self.conv3 = nn.Conv2d(64, 64, kernel=1)
        self.conv4 = nn.Conv2d(64, 64, kernel=1)
        self.conv5 = nn.Conv2d(64, 64, kernel=1)
        self.conv6 = nn.Conv2d(64, 64, kernel=1)
        self.conv7 = nn.Conv2d(64, 64, kernel=1)
        self.conv8 = nn.Conv2d(64, 64, kernel=1)
        self.conv9 = nn.Conv2d(64, 64, kernel=1)
```