Coursework 2: Image segmentation

In this coursework you will develop and train a convolutional neural network for brain tumour image segmentation. Please read both the text and the code in this notebook to get an idea what you are expected to implement. Pay attention to the missing code blocks that look like this:

```
### Insert your code ###
...
### End of your code ###
```

What to do?

- Complete and run the code using `jupyter-lab` or `jupyter-notebook` to get the results.
- Export (File | Save and Export Notebook As...) the notebook as a PDF file, which contains your code, results and answers, and upload the PDF file onto Scientia (https://scientia.doc.ic.ac.uk).
- Instead of clicking the Export button, you can also run the following command instead: `jupyter nbconvert coursework.ipynb --to pdf`
- If Jupyter complains about some problems in exporting, it is likely that pandoc (https://pandoc.org/installing.html (https://pandoc.org/installing.html)) or latex is not installed, or their paths have not been included. You can install the relevant libraries and retry.
- If Jupyter-lab does not work for you at the end, you can use Google Colab to write the code and export the PDF file.

Dependencies

You need to install Jupyter-Lab (https://jupyterlab.readthedocs.io/en/stable/getting_started/installation.html (https://jupyterlab.readthedocs.io/en/stable/getting_started/installation.html)) and other libraries used in this coursework, such as by running the command: `pip3 install [package_name]`

GPU resource

The coursework is developed to be able to run on CPU, as all images have been pre-processed to be 2D and of a smaller size, compared to original 3D volumes.

However, to save training time, you may want to use GPU. In that case, you can run this notebook on Google Colab. On Google Colab, go to the menu, Runtime - Change runtime type, and select **GPU** as the hardware acceleartor. At the end, please still export everything and submit as a PDF file on Scientia.

In [1]:

```
# Import libraries
# These libraries should be sufficient for this tutorial.
# However, if any other library is needed, please install by yourself.
import tarfile
import imageio
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset
import numpy as np
import time
import os
import random
import matplotlib.pyplot as plt
from matplotlib import colors
```

1. Download and visualise the imaging dataset.

The dataset is curated from the brain imaging dataset in Medical Decathlon Challenge (http://medicaldecathlon.com/). To save the storage and reduce the computational cost for this tutorial, we extract 2D image slices from T1-Gd contrast enhanced 3D brain volumes and downsample the images.

The dataset consists of a training set and a test set. Each image is of dimension 120 x 120, with a corresponding label map of the same dimension. There are four number of classes in the label map:

- 0: background
- 1: edema
- 2: non-enhancing tumour
- 3: enhancing tumour

In [2]:

```
# Download the dataset
!wget https://www.dropbox.com/s/zmytk2yu284af6t/Task01_BrainTumour_2D.tar.gz

# Unzip the '.tar.gz' file to the current directory
datafile = tarfile.open('Task01_BrainTumour_2D.tar.gz')
datafile.extractall()
datafile.close()
```

```
--2023-03-03 13:17:14--  https://www.dropbox.com/s/zmytk2yu284af6t/Tas
k01_BrainTumour_2D.tar.gz (https://www.dropbox.com/s/zmytk2yu284af6t/T
ask01_BrainTumour_2D.tar.gz)
Resolving www.dropbox.com (www.dropbox.com)... 162.125.1.18, 2620:100:
6016:18::a27d:112
Connecting to www.dropbox.com (www.dropbox.com)|162.125.1.18|:443... c
onnected.
HTTP request sent, awaiting response... 302 Found
Location: /s/raw/zmytk2yu284af6t/Task01_BrainTumour_2D.tar.gz [followi
ng]
--2023-03-03 13:17:15--  https://www.dropbox.com/s/raw/zmytk2yu284af6
t/Task01_BrainTumour_2D.tar.gz (https://www.dropbox.com/s/raw/zmytk2yu
284af6t/Task01_BrainTumour_2D.tar.gz)
Reusing existing connection to www.dropbox.com:443.
HTTP request sent, awaiting response... 302 Found
Location: https://uc04907ead56e5d8bd17897f5bed.dl.dropboxusercontent.c
om/cd/0/inline/B3iC93bW2LeM7a2IOn30gBpzXkS2HAHiw_6Cler7GYgv-1rDajmK4RX
aosliY692eNydDje5L9sYhEuRWLaPBhwmchRJUPkUkSHhe3P2p2XeMwRKlhaWLur95HML-
yAHndhF8PnOUT-aIesswSuSIKabDJy-dDDXoD6vWVwB9v5kFg/file# (https://uc049
07ead56e5d8bd17897f5bed.dl.dropboxusercontent.com/cd/0/inline/B3iC93bW
2LeM7a2IOn30gBpzXkS2HAHiw_6Cler7GYgv-1rDajmK4RXaosliY692eNydDje5L9sYhE
uRWLaPBhwmchRJUPkUkSHhe3P2p2XeMwRKlhaWLur95HML-yAHndhF8PnOUT-aIesswSuS
IKabDJy-dDDXoD6vWVwB9v5kFg/file#) [following]
--2023-03-03 13:17:15--  https://uc04907ead56e5d8bd17897f5bed.dl.dropb
oxusercontent.com/cd/0/inline/B3iC93bW2LeM7a2IOn30gBpzXkS2HAHiw_6Cler7
GYgv-1rDajmK4RXaosliY692eNydDje5L9sYhEuRWLaPBhwmchRJUPkUkSHhe3P2p2XeMw
RKlhaWLur95HML-yAHndhF8PnOUT-aIesswSuSIKabDJy-dDDXoD6vWVwB9v5kFg/file
(https://uc04907ead56e5d8bd17897f5bed.dl.dropboxusercontent.com/cd/0/i
nline/B3iC93bW2LeM7a2IOn30gBpzXkS2HAHiw_6Cler7GYgv-1rDajmK4RXaosliY692
eNydDje5L9sYhEuRWLaPBhwmchRJUPkUkSHhe3P2p2XeMwRKlhaWLur95HML-yAHndhF8P
nOUT-aIesswSuSIKabDJy-dDDXoD6vWVwB9v5kFg/file)
Resolving uc04907ead56e5d8bd17897f5bed.dl.dropboxusercontent.com (uc04
907ead56e5d8bd17897f5bed.dl.dropboxusercontent.com)... 162.125.3.15, 2
620:100:6016:15::a27d:10f
Connecting to uc04907ead56e5d8bd17897f5bed.dl.dropboxusercontent.com
(uc04907ead56e5d8bd17897f5bed.dl.dropboxusercontent.com)|162.125.3.15
|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: /cd/0/inline2/B3hSRiIcx0wADdFSmDYAINdSJ7q0FwA0T3p-sj2XXU8eL6
IzXFOpgMlX9hDFcn5EGKfifZFtwiTbUPUtZZ8HAAvHvEETEf-n9VfgmVO6zVuucvH-M2Gf
2LBVxAvtIlvlVgrd7Wei14i-qsJ1WgsUcrouF5BxkSDpgJnLIiLFoSlpLwZbn374XANn2S
ljBocgMqM7-Q3U6RyUrmJnUXYYgeaY973UKpA4OH74euw72fttqxB3LJSmmJ5c4Z2bhcca
m0o8i6h3Wmsu54iersx265yCMKNbeK5Z0sVWxyN6O2RRm0AIfnUSVdaGVkG5ZA1cYWN-tk
6gZnDh3npTtLY3fCu5NVPibFCbpod2XJvRMfpipJaZIrPuR_R4Jzp3AXoo819NDfnlIBl0
iUpU0DqXmM9vyj2MRaxCQ-jB040bOAgdZw/file [following]
--2023-03-03 13:17:16--  https://uc04907ead56e5d8bd17897f5bed.dl.dropb
oxusercontent.com/cd/0/inline2/B3hSRiIcx0wADdFSmDYAINdSJ7q0FwA0T3p-sj2
XXU8eL6IzXFOpgMlX9hDFcn5EGKfifZFtwiTbUPUtZZ8HAAvHvEETEf-n9VfgmVO6zVuuc
vH-M2Gf2LBVxAvtIlvlVgrd7Wei14i-qsJ1WgsUcrouF5BxkSDpgJnLIiLFoSlpLwZbn37
4XANn2SljBocgMqM7-Q3U6RyUrmJnUXYYgeaY973UKpA4OH74euw72fttqxB3LJSmmJ5c4
Z2bhccam0o8i6h3Wmsu54iersx265yCMKNbeK5Z0sVWxyN6O2RRm0AIfnUSVdaGVkG5ZA1
cYWN-tk6gZnDh3npTtLY3fCu5NVPibFCbpod2XJvRMfpipJaZIrPuR_R4Jzp3AXoo819ND
fnlIBl0iUpU0DqXmM9vyj2MRaxCQ-jB040bOAgdZw/file (https://uc04907ead56e5
d8bd17897f5bed.dl.dropboxusercontent.com/cd/0/inline2/B3hSRiIcx0wADdFS
mDYAINdSJ7q0FwA0T3p-sj2XXU8eL6IzXFOpgMlX9hDFcn5EGKfifZFtwiTbUPUtZZ8HAA
vHvEETEf-n9VfgmVO6zVuucvH-M2Gf2LBVxAvtIlvlVgrd7Wei14i-qsJ1WgsUcrouF5Bx
kSDpgJnLIiLFoSlpLwZbn374XANn2SljBocgMqM7-Q3U6RyUrmJnUXYYgeaY973UKpA4OH
74euw72fttqxB3LJSmmJ5c4Z2bhccam0o8i6h3Wmsu54iersx265yCMKNbeK5Z0sVWxyN6
O2RRm0AIfnUSVdaGVkG5ZA1cYWN-tk6gZnDh3npTtLY3fCu5NVPibFCbpod2XJvRMfpipJ
aZIrPuR_R4Jzp3AXoo819NDfnlIBl0iUpU0DqXmM9vyj2MRaxCQ-jB040bOAgdZw/file)
Reusing existing connection to uc04907ead56e5d8bd17897f5bed.dl.dropbox
```

```
usercontent.com:443.
HTTP request sent, awaiting response... 200 OK
Length: 9251149 (8.8M) [application/octet-stream]
Saving to: ‘Task01_BrainTumour_2D.tar.gz’

Task01_BrainTumour_ 100%[===================>]   8.82M  21.8MB/s    in
0.4s

2023-03-03 13:17:17 (21.8 MB/s) - ‘Task01_BrainTumour_2D.tar.gz’ saved
[9251149/9251149]
```

Visualise a random set of 4 training images along with their label maps.

Suggested colour map for brain MR image:

```
cmap = 'gray'
```

Suggested colour map for segmentation map:

```
cmap = colors.ListedColormap(['black', 'green', 'blue', 'red'])
```

In [3]:

```
# Define paths for training images and labels
train_image_path = "Task01_BrainTumour_2D/training_images"
train_label_path = "Task01_BrainTumour_2D/training_labels"

image_files = os.listdir(train_image_path)
label_files = os.listdir(train_label_path)

# Define colour maps for image and label
image_cmap = "gray"
label_cmap = colors.ListedColormap(["black", "green", "blue", "red"])

plt.figure(figsize=(10, 10))

# Loop over 4 random indices
for i in range(4):
    index = random.randint(0, len(image_files) - 1)
    image_file = image_files[index]
    label_file = label_files[index]

    # Read the images
    image = plt.imread(os.path.join(train_image_path, image_file))
    label = plt.imread(os.path.join(train_label_path, label_file))

    # Plot the images
    plt.subplot(4, 2, i * 2 + 1)
    plt.imshow(image, cmap=image_cmap)
    plt.title(f"Image: {image_file}")
    plt.axis("off")


    plt.subplot(4, 2, i * 2 + 2)
    plt.imshow(label, cmap=label_cmap)
    plt.title(f"Label: {label_file}")
    plt.axis("off")

# Show the figure
plt.show()
```

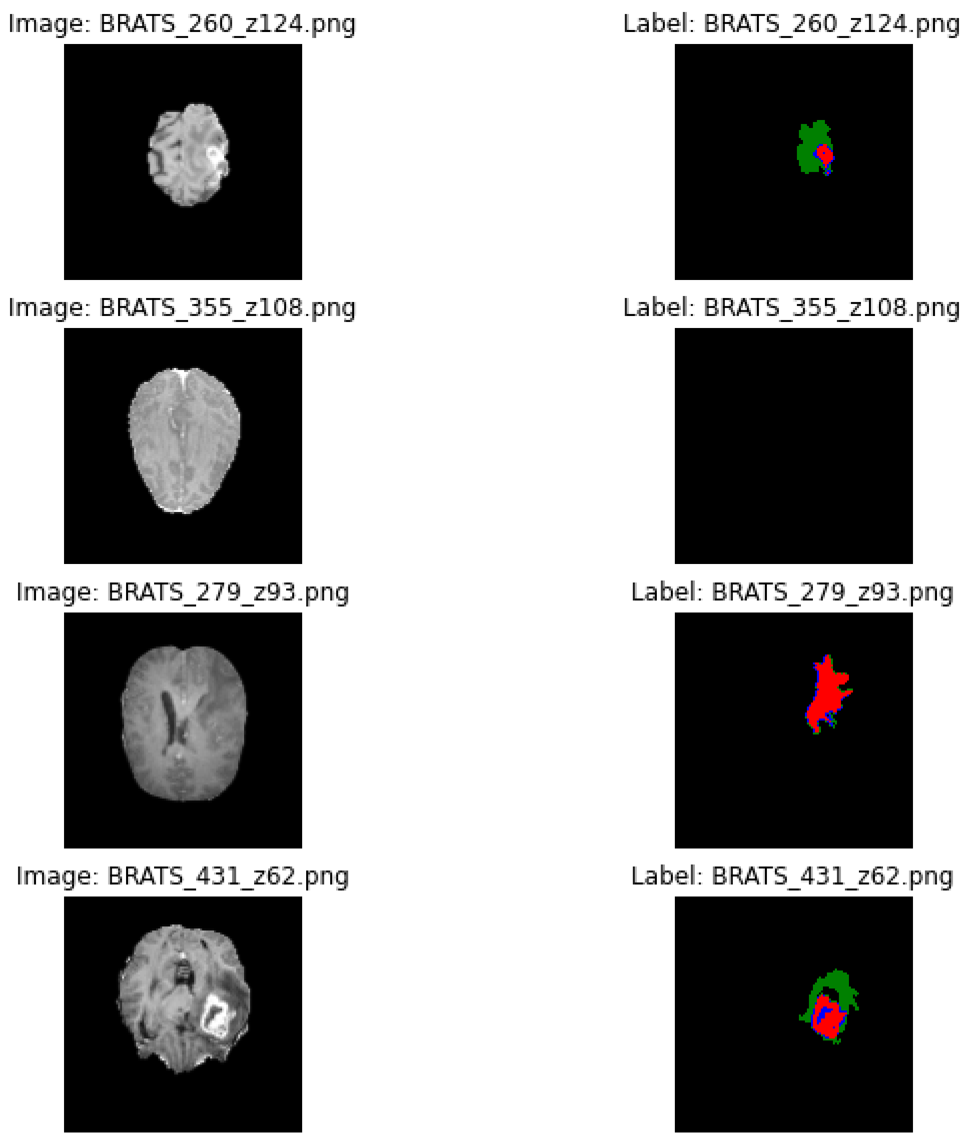


Image: BRATS_260_z124.png
Label: BRATS_260_z124.png
Image: BRATS_355_z108.png
Label: BRATS_355_z108.png
Image: BRATS_279_z93.png
Label: BRATS_279_z93.png
Image: BRATS_431_z62.png
Label: BRATS_431_z62.png

2. Implement a dataset class.

It can read the imaging dataset and get items, pairs of images and label maps, as training batches.

In [4]:

```
def normalise_intensity(image, thres_roi=1.0):
    """ Normalise the image intensity by the mean and standard deviation """
    # ROI defines the image foreground
    val_l = np.percentile(image, thres_roi)
    roi = (image >= val_l)
    mu, sigma = np.mean(image[roi]), np.std(image[roi])
    eps = 1e-6
    image2 = (image - mu) / (sigma + eps)
    return image2


class BrainImageSet(Dataset):
    """ Brain image set """
    def __init__(self, image_path, label_path='', deploy=False):
        self.image_path = image_path
        self.deploy = deploy
        self.images = []
        self.labels = []

        image_names = sorted(os.listdir(image_path))
        for image_name in image_names:
            # Read the image
            image = imageio.imread(os.path.join(image_path, image_name))
            self.images += [image]

            # Read the label map
            if not self.deploy:
                label_name = os.path.join(label_path, image_name)
                label = imageio.imread(label_name)
                self.labels += [label]

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

    def __getitem__(self, idx):
        # Get an image and perform intensity normalisation
        # Dimension: XY
        image = normalise_intensity(self.images[idx])

        # Get its label map
        # Dimension: XY
        label = self.labels[idx]
        return image, label

    def get_random_batch(self, batch_size):
        # Get a batch of paired images and label maps
        # Dimension of images: NCXY
        # Dimension of labels: NXY
        images, labels = [], []

        ### Insert your code ###################################################
        # randomly select number of images
        indices = random.sample(range(len(self)), batch_size)

        for idx in indices:
            # perform intensity normalisation
            # Dimension is normal: XY
            image = normalise_intensity(self.images[idx])
            # Dimension: 1X1XY
            image = np.expand_dims(np.expand_dims(image, axis=0), axis=0)

            # Get its label map and add a channel dimension
            # Dimension: 1XY
            label = self.labels[idx]
            label = np.expand_dims(label, axis=0)

            # Add the image and label to the lists
            images.append(image)
            labels.append(label)

        # Stack the images and labels to create a 4D tensor
        images = np.vstack(images)
        labels = np.vstack(labels)
        ### End of your code ##################################################
        return images, labels
```

In [5]:

```
# Testing part 2
dataset = BrainImageSet(image_path='Task01_BrainTumour_2D/training_images',
                        label_path='Task01_BrainTumour_2D/training_labels')

image, label = dataset[0]
print(f"Image shape: {image.shape}, Label shape: {label.shape}")

batch_size = 4
images, labels = dataset.get_random_batch(batch_size)
print(f"Images shape: {images.shape}, Labels shape: {labels.shape}")
```

```
Image shape: (120, 120), Label shape: (120, 120)
Images shape: (4, 1, 120, 120), Labels shape: (4, 120, 120)
```

3. Build a U-net architecture.

You will implement a U-net architecture. If you are not familiar with U-net, please read this paper:

[1] Olaf Ronneberger et al. U-Net: Convolutional networks for biomedical image segmentation (https://link.springer.com/chapter/10.1007/978-3-319-24574-4_28). MICCAI, 2015.

For the first convolutional layer, you can start with 16 filters. We have implemented the encoder path. Please complete the decoder path.

In [6]:

```
""" U-net """
class UNet(nn.Module):
    def __init__(self, input_channel=1, output_channel=1, num_filter=16):
        super(UNet, self).__init__()

        # BatchNorm: by default during training this layer keeps running estimates
        # of its computed mean and variance, which are then used for normalization
        # during evaluation.

        # Encoder path
        n = num_filter  # 16
        self.conv1 = nn.Sequential(
            nn.Conv2d(input_channel, n, kernel_size=3, padding=1),
            nn.BatchNorm2d(n),
            nn.ReLU(),
            nn.Conv2d(n, n, kernel_size=3, padding=1),
            nn.BatchNorm2d(n),
            nn.ReLU()
        )

        n *= 2  # 32
        self.conv2 = nn.Sequential(
            nn.Conv2d(int(n / 2), n, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(n),
            nn.ReLU(),
            nn.Conv2d(n, n, kernel_size=3, padding=1),
            nn.BatchNorm2d(n),
            nn.ReLU()
        )

        n *= 2  # 64
        self.conv3 = nn.Sequential(
            nn.Conv2d(int(n / 2), n, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(n),
            nn.ReLU(),
            nn.Conv2d(n, n, kernel_size=3, padding=1),
            nn.BatchNorm2d(n),
            nn.ReLU()
        )

        n *= 2  # 128
        self.conv4 = nn.Sequential(
            nn.Conv2d(int(n / 2), n, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(n),
            nn.ReLU(),
            nn.Conv2d(n, n, kernel_size=3, padding=1),
            nn.BatchNorm2d(n),
            nn.ReLU()
        )

        # Decoder path
        ### Insert your code

        # Bottom layer

        self.inter = nn.Sequential(
            nn.Conv2d(n, 2*n, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(2*n),
            nn.ReLU(),
            nn.ConvTranspose2d(2*n, n, kernel_size=3, dilation = 3, output_padding=1
            nn.BatchNorm2d(n),
            nn.ReLU()
        )


        self.conv_up4 = nn.Sequential(
            nn.Conv2d(2*n, n, kernel_size=3, padding=1),
            nn.BatchNorm2d(n),
            nn.ReLU(),
            nn.ConvTranspose2d(n, int(n/2), kernel_size=3, stride=2, padding=1, outp
            nn.BatchNorm2d(int(n/2)),
            nn.ReLU()
        )

        n //= 2  # 64
        self.conv_up3 = nn.Sequential(
            nn.Conv2d(2*n, n, kernel_size=3, padding=1),
            nn.BatchNorm2d(n),
            nn.ReLU(),
            nn.ConvTranspose2d(n, int(n/2), kernel_size=3, stride=2, padding=1, outp
            nn.BatchNorm2d(int(n/2)),
            nn.ReLU()
        )

        n //= 2  # 32
        self.conv_up2 = nn.Sequential(
            nn.Conv2d(2*n, n, kernel_size=3, padding=1),
            nn.BatchNorm2d(n),
            nn.ReLU(),
            nn.ConvTranspose2d(n, int(n/2), kernel_size=3, stride=2, padding=1, outp
            nn.BatchNorm2d(int(n/2)),
            nn.ReLU()
        )
        n //= 2  # 16
        self.conv_up1 = nn.Sequential(
            nn.Conv2d(n*2, n, kernel_size=3, padding=1),
            nn.BatchNorm2d(n),
            nn.ReLU(),
            nn.Conv2d(n, output_channel, kernel_size=1)
        )


                 # 64
        # self.conv_up4 = nn.Sequential(
        #     nn.ConvTranspose2d(n, int(n / 2), kernel_size=3, stride=2, padding=1,
        #     nn.BatchNorm2d(int(n / 2)),
        #     nn.ReLU(),
        #     nn.Conv2d(int(n / 2), int(n / 2), kernel_size=3, padding=1),
        #     nn.BatchNorm2d(int(n / 2)),
        #     nn.ReLU()
        # )

        ### End of your code

    def forward(self, x):
        # Use the convolutional operators defined above to build the U-net
        # The encoder part is already done for you.
        # You need to complete the decoder part.
        # Encoder
        x = self.conv1(x)
        conv1_skip = x

        x = self.conv2(x)
        conv2_skip = x

        x = self.conv3(x)
        conv3_skip = x

        x = self.conv4(x)
        conv4_skip = x

        x = self.inter(x)


        # Decoder
        ### Insert your code


        x = torch.cat([x, conv4_skip], 1)

        x = self.conv_up4(x)

        x = torch.cat([x, conv3_skip], 1)

        x = self.conv_up3(x)

        x = torch.cat([x, conv2_skip], 1)

        x = self.conv_up2(x)

        x = torch.cat([x, conv1_skip], 1)

        x = self.conv_up1(x)

        ### End of your code
        return x
```

In [7]:

```
# Test part 3
from torchsummary import summary

# CUDA device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print('Device: {0}'.format(device))

# Build the model
num_class = 4
model = UNet(input_channel=1, output_channel=num_class, num_filter=16)
model = model.to(device)
params = list(model.parameters())

print(summary(model, (1, 120, 120)))
```

```
Device: cuda
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 16, 120, 120]             160
       BatchNorm2d-2         [-1, 16, 120, 120]              32
              ReLU-3         [-1, 16, 120, 120]               0
            Conv2d-4         [-1, 16, 120, 120]           2,320
       BatchNorm2d-5         [-1, 16, 120, 120]              32
              ReLU-6         [-1, 16, 120, 120]               0
            Conv2d-7           [-1, 32, 60, 60]           4,640
       BatchNorm2d-8           [-1, 32, 60, 60]              64
              ReLU-9           [-1, 32, 60, 60]               0
           Conv2d-10           [-1, 32, 60, 60]           9,248
      BatchNorm2d-11           [-1, 32, 60, 60]              64
             ReLU-12           [-1, 32, 60, 60]               0
           Conv2d-13           [-1, 64, 30, 30]          18,496
      BatchNorm2d-14           [-1, 64, 30, 30]             128
             ReLU-15           [-1, 64, 30, 30]               0
           Conv2d-16           [-1, 64, 30, 30]          36,928
      BatchNorm2d-17           [-1, 64, 30, 30]             128
             ReLU-18           [-1, 64, 30, 30]               0
           Conv2d-19          [-1, 128, 15, 15]          73,856
      BatchNorm2d-20          [-1, 128, 15, 15]             256
             ReLU-21          [-1, 128, 15, 15]               0
           Conv2d-22          [-1, 128, 15, 15]         147,584
      BatchNorm2d-23          [-1, 128, 15, 15]             256
             ReLU-24          [-1, 128, 15, 15]               0
           Conv2d-25            [-1, 256, 8, 8]         295,168
      BatchNorm2d-26            [-1, 256, 8, 8]             512
             ReLU-27            [-1, 256, 8, 8]               0
  ConvTranspose2d-28          [-1, 128, 15, 15]         295,040
      BatchNorm2d-29          [-1, 128, 15, 15]             256
             ReLU-30          [-1, 128, 15, 15]               0
           Conv2d-31          [-1, 128, 15, 15]         295,040
      BatchNorm2d-32          [-1, 128, 15, 15]             256
             ReLU-33          [-1, 128, 15, 15]               0
  ConvTranspose2d-34           [-1, 64, 30, 30]          73,792
      BatchNorm2d-35           [-1, 64, 30, 30]             128
             ReLU-36           [-1, 64, 30, 30]               0
           Conv2d-37           [-1, 64, 30, 30]          73,792
      BatchNorm2d-38           [-1, 64, 30, 30]             128
             ReLU-39           [-1, 64, 30, 30]               0
  ConvTranspose2d-40           [-1, 32, 60, 60]          18,464
      BatchNorm2d-41           [-1, 32, 60, 60]              64
             ReLU-42           [-1, 32, 60, 60]               0
           Conv2d-43           [-1, 32, 60, 60]          18,464
      BatchNorm2d-44           [-1, 32, 60, 60]              64
             ReLU-45           [-1, 32, 60, 60]               0
  ConvTranspose2d-46         [-1, 16, 120, 120]           4,624
      BatchNorm2d-47         [-1, 16, 120, 120]              32
             ReLU-48         [-1, 16, 120, 120]               0
           Conv2d-49         [-1, 16, 120, 120]           4,624
      BatchNorm2d-50         [-1, 16, 120, 120]              32
             ReLU-51         [-1, 16, 120, 120]               0
           Conv2d-52          [-1, 4, 120, 120]              68
================================================================
Total params: 1,374,740
Trainable params: 1,374,740
Non-trainable params: 0
----------------------------------------------------------------
```

```
Input size (MB): 0.05
Forward/backward pass size (MB): 40.37
Params size (MB): 5.24
Estimated Total Size (MB): 45.66
----------------------------------------------------------------
None
```

4. Train the segmentation model.

In [8]:

```
# CUDA device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print('Device: {0}'.format(device))

# Build the model
num_class = 4
model = UNet(input_channel=1, output_channel=num_class, num_filter=16)
model = model.to(device)
params = list(model.parameters())

model_dir = 'saved_models'
if not os.path.exists(model_dir):
    os.makedirs(model_dir)

# Optimizer
optimizer = optim.Adam(params, lr=1e-3)

# Segmentation loss
criterion = nn.CrossEntropyLoss()

# Datasets
train_set = BrainImageSet('Task01_BrainTumour_2D/training_images', 'Task01_BrainTumo
test_set = BrainImageSet('Task01_BrainTumour_2D/test_images', 'Task01_BrainTumour_2D

# Train the model
# Note: when you debug the model, you may reduce the number of iterations or batch s
# CHANGE THIS BACK TO BIGGGG
num_iter = 10000
train_batch_size = 16
eval_batch_size = 16
start = time.time()
for it in range(1, 1 + num_iter):
    # Set the modules in training mode, which will have effects on certain modules,
    start_iter = time.time()
    model.train()

    # Get a batch of images and labels
    images, labels = train_set.get_random_batch(train_batch_size)
    images, labels = torch.from_numpy(images), torch.from_numpy(labels)
    images, labels = images.to(device, dtype=torch.float32), labels.to(device, dtype
    logits = model(images)

    # Perform optimisation and print out the training loss
    ### Insert your code ###

    loss = criterion(logits, labels)
    optimizer.zero_grad()

    loss.backward()
    optimizer.step()
    ### End of your code ###

    # Evaluate
    if it % 100 == 0:
        model.eval()
        # Disabling gradient calculation during reference to reduce memory consumpti
        with torch.no_grad():
            # Evaluate on a batch of test images and print out the test loss
            ### Insert your code ###
            total_loss = 0
            images, labels = test_set.get_random_batch(eval_batch_size)
            images, labels = torch.from_numpy(images), torch.from_numpy(labels)
            images, labels = images.to(device, dtype=torch.float32), labels.to(devic
            logits = model(images)
            loss = criterion(logits, labels)
            total_loss += loss.item()
            avg_loss = total_loss / (len(test_set) / eval_batch_size)
            print('Iteration [{0}/{1}], test loss: {2:.4f}, loss: {3:.4f}'\
            .format(it, num_iter, avg_loss, loss))
            ### End of your code ###

    # Save the model
    if it % 5000 == 0:
        torch.save(model.state_dict(), os.path.join(model_dir, 'model_{0}.pt'.format
print('Training took {:.3f}s in total.'.format(time.time() - start))
```

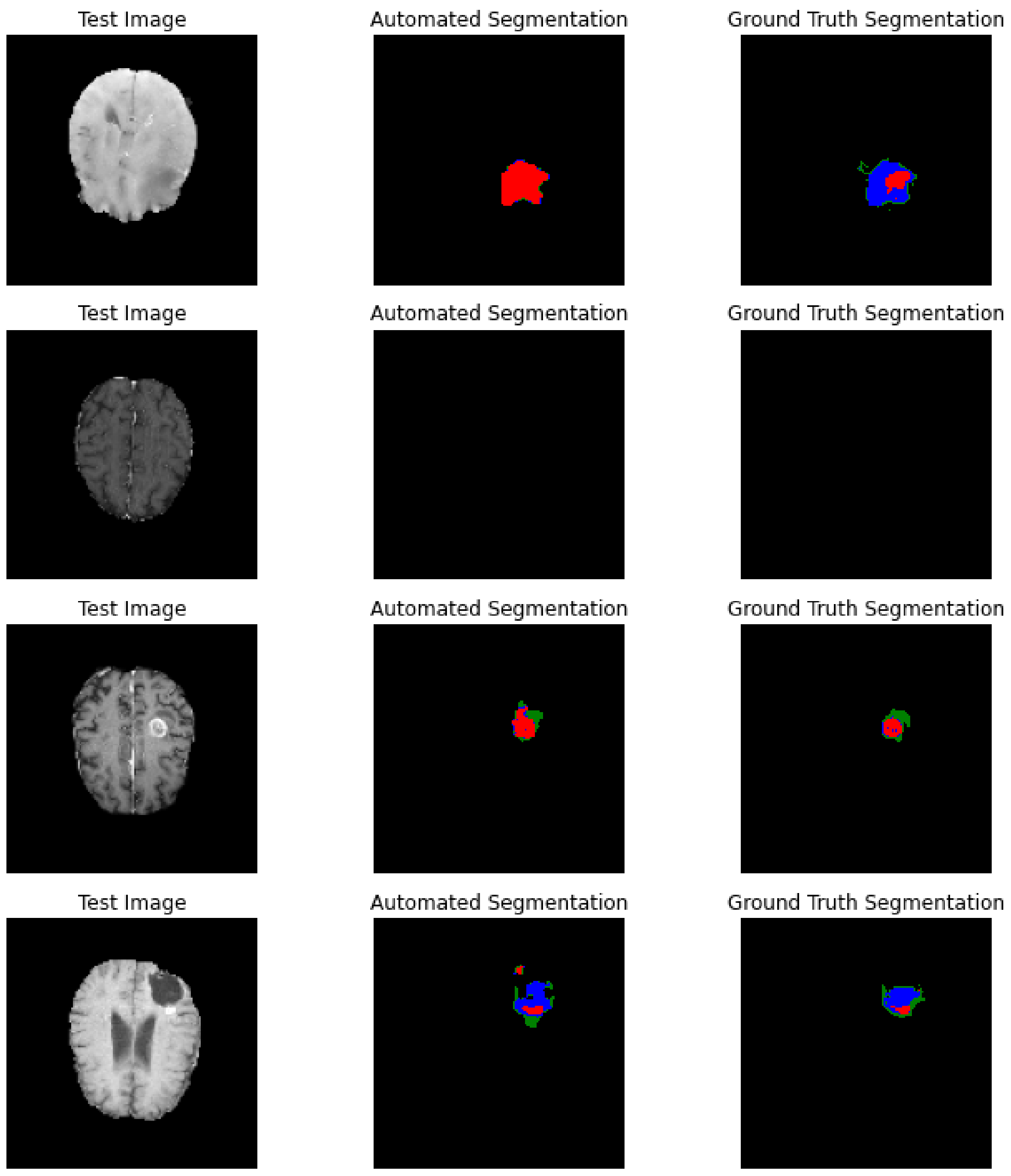
```
Device: cuda
Iteration [100/10000], test loss: 0.0068, loss: 0.3119
Iteration [200/10000], test loss: 0.0027, loss: 0.1213
Iteration [300/10000], test loss: 0.0023, loss: 0.1061
Iteration [400/10000], test loss: 0.0016, loss: 0.0723
Iteration [500/10000], test loss: 0.0018, loss: 0.0831
Iteration [600/10000], test loss: 0.0014, loss: 0.0629
Iteration [700/10000], test loss: 0.0015, loss: 0.0692
Iteration [800/10000], test loss: 0.0015, loss: 0.0677
Iteration [900/10000], test loss: 0.0014, loss: 0.0646
Iteration [1000/10000], test loss: 0.0012, loss: 0.0567
Iteration [1100/10000], test loss: 0.0012, loss: 0.0558
Iteration [1200/10000], test loss: 0.0008, loss: 0.0359
Iteration [1300/10000], test loss: 0.0011, loss: 0.0524
Iteration [1400/10000], test loss: 0.0008, loss: 0.0351
Iteration [1500/10000], test loss: 0.0010, loss: 0.0463
Iteration [1600/10000], test loss: 0.0006, loss: 0.0256
Iteration [1700/10000], test loss: 0.0008, loss: 0.0349
Iteration [1800/10000], test loss: 0.0009, loss: 0.0422
Iteration [1900/10000], test loss: 0.0007, loss: 0.0327
Iteration [2000/10000], test loss: 0.0006, loss: 0.0290
Iteration [2100/10000], test loss: 0.0010, loss: 0.0457
Iteration [2200/10000], test loss: 0.0007, loss: 0.0340
Iteration [2300/10000], test loss: 0.0008, loss: 0.0362
Iteration [2400/10000], test loss: 0.0010, loss: 0.0444
Iteration [2500/10000], test loss: 0.0006, loss: 0.0292
Iteration [2600/10000], test loss: 0.0006, loss: 0.0260
Iteration [2700/10000], test loss: 0.0007, loss: 0.0335
Iteration [2800/10000], test loss: 0.0009, loss: 0.0408
Iteration [2900/10000], test loss: 0.0005, loss: 0.0234
Iteration [3000/10000], test loss: 0.0009, loss: 0.0412
Iteration [3100/10000], test loss: 0.0006, loss: 0.0253
Iteration [3200/10000], test loss: 0.0008, loss: 0.0355
Iteration [3300/10000], test loss: 0.0008, loss: 0.0377
Iteration [3400/10000], test loss: 0.0006, loss: 0.0266
Iteration [3500/10000], test loss: 0.0011, loss: 0.0487
Iteration [3600/10000], test loss: 0.0008, loss: 0.0384
Iteration [3700/10000], test loss: 0.0007, loss: 0.0304
Iteration [3800/10000], test loss: 0.0007, loss: 0.0323
Iteration [3900/10000], test loss: 0.0007, loss: 0.0315
Iteration [4000/10000], test loss: 0.0005, loss: 0.0206
Iteration [4100/10000], test loss: 0.0006, loss: 0.0296
Iteration [4200/10000], test loss: 0.0005, loss: 0.0224
Iteration [4300/10000], test loss: 0.0012, loss: 0.0565
Iteration [4400/10000], test loss: 0.0006, loss: 0.0268
Iteration [4500/10000], test loss: 0.0008, loss: 0.0358
Iteration [4600/10000], test loss: 0.0007, loss: 0.0319
Iteration [4700/10000], test loss: 0.0007, loss: 0.0303
Iteration [4800/10000], test loss: 0.0007, loss: 0.0322
Iteration [4900/10000], test loss: 0.0009, loss: 0.0395
Iteration [5000/10000], test loss: 0.0012, loss: 0.0544
Iteration [5100/10000], test loss: 0.0007, loss: 0.0302
Iteration [5200/10000], test loss: 0.0005, loss: 0.0215
Iteration [5300/10000], test loss: 0.0009, loss: 0.0422
Iteration [5400/10000], test loss: 0.0009, loss: 0.0393
Iteration [5500/10000], test loss: 0.0011, loss: 0.0510
Iteration [5600/10000], test loss: 0.0006, loss: 0.0292
Iteration [5700/10000], test loss: 0.0008, loss: 0.0377
Iteration [5800/10000], test loss: 0.0005, loss: 0.0213
Iteration [5900/10000], test loss: 0.0005, loss: 0.0210
Iteration [6000/10000], test loss: 0.0007, loss: 0.0341
Iteration [6100/10000], test loss: 0.0007, loss: 0.0336
Iteration [6200/10000], test loss: 0.0008, loss: 0.0353
Iteration [6300/10000], test loss: 0.0011, loss: 0.0519
Iteration [6400/10000], test loss: 0.0009, loss: 0.0425
Iteration [6500/10000], test loss: 0.0013, loss: 0.0571
Iteration [6600/10000], test loss: 0.0008, loss: 0.0352
Iteration [6700/10000], test loss: 0.0005, loss: 0.0231
Iteration [6800/10000], test loss: 0.0011, loss: 0.0502
Iteration [6900/10000], test loss: 0.0013, loss: 0.0596
Iteration [7000/10000], test loss: 0.0010, loss: 0.0440
Iteration [7100/10000], test loss: 0.0008, loss: 0.0342
Iteration [7200/10000], test loss: 0.0009, loss: 0.0418
Iteration [7300/10000], test loss: 0.0013, loss: 0.0612
Iteration [7400/10000], test loss: 0.0004, loss: 0.0177
Iteration [7500/10000], test loss: 0.0009, loss: 0.0395
Iteration [7600/10000], test loss: 0.0010, loss: 0.0454
Iteration [7700/10000], test loss: 0.0011, loss: 0.0487
Iteration [7800/10000], test loss: 0.0014, loss: 0.0661
Iteration [7900/10000], test loss: 0.0009, loss: 0.0430
Iteration [8000/10000], test loss: 0.0010, loss: 0.0468
Iteration [8100/10000], test loss: 0.0013, loss: 0.0578
Iteration [8200/10000], test loss: 0.0010, loss: 0.0454
Iteration [8300/10000], test loss: 0.0009, loss: 0.0425
Iteration [8400/10000], test loss: 0.0008, loss: 0.0380
Iteration [8500/10000], test loss: 0.0009, loss: 0.0426
Iteration [8600/10000], test loss: 0.0006, loss: 0.0273
Iteration [8700/10000], test loss: 0.0010, loss: 0.0435
Iteration [8800/10000], test loss: 0.0009, loss: 0.0405
Iteration [8900/10000], test loss: 0.0008, loss: 0.0377
Iteration [9000/10000], test loss: 0.0005, loss: 0.0232
Iteration [9100/10000], test loss: 0.0013, loss: 0.0586
Iteration [9200/10000], test loss: 0.0020, loss: 0.0903
Iteration [9300/10000], test loss: 0.0009, loss: 0.0391
Iteration [9400/10000], test loss: 0.0010, loss: 0.0464
Iteration [9500/10000], test loss: 0.0008, loss: 0.0387
Iteration [9600/10000], test loss: 0.0005, loss: 0.0242
Iteration [9700/10000], test loss: 0.0009, loss: 0.0394
Iteration [9800/10000], test loss: 0.0005, loss: 0.0223
Iteration [9900/10000], test loss: 0.0016, loss: 0.0712
Iteration [10000/10000], test loss: 0.0019, loss: 0.0889
Training took 359.858s in total.
```

5. Deploy the trained model to a random set of 4 test images and visualise the automated segmentation.

You can show the images as a 4 x 3 panel. Each row shows one example, with the 3 columns being the test image, automated segmentation and ground truth segmentation.

In [23]:

```
### Insert your code ###

label_cmap = colors.ListedColormap(["black", "green", "blue", "red"])

# get test images and labels
test_images, test_labels = test_set.get_random_batch(4)

# use evaluation mode
model.eval()

# get the automated segmentation for each test image using model
test_images = torch.from_numpy(test_images).to(device, dtype=torch.float32)
with torch.no_grad():
    test_logits = model(test_images)
test_segmentations = torch.argmax(test_logits, dim=1).cpu().numpy()

num_test_images = 4
fig, axs = plt.subplots(nrows=num_test_images, ncols=3, figsize=(10, 10))

for i in range(num_test_images):
    # test image
    axs[i, 0].imshow(test_images[i, 0, :, :].cpu(), cmap='gray')
    axs[i, 0].axis('off')
    axs[i, 0].set_title('Test Image')

    # automated segmentation
    axs[i, 1].imshow(test_segmentations[i, :, :], cmap=label_cmap)
    axs[i, 1].axis('off')
    axs[i, 1].set_title('Automated Segmentation')

    # ground truth segmentation
    axs[i, 2].imshow(test_labels[i, :, :], cmap=label_cmap)
    axs[i, 2].axis('off')
    axs[i, 2].set_title('Ground Truth Segmentation')

plt.tight_layout()
plt.show()

### End of your code ###
```



6. Discussion. Does your trained model work well? How would you improve this model so it can be deployed to the real clinic?

To evaluate the performance of the trained model, various evaluation metrics can be computed, such as sensitivity, specificity, precision, and others. These metrics are crucial in determining the model's efficacy, as they directly impact patient outcomes. A model with high sensitivity can accurately detect a larger proportion of true positive cases, such as tumors, ensuring timely and appropriate treatment. Similarly, a model with high specificity can minimize false positive cases, avoiding unnecessary interventions or misdiagnosis.

One important technique that can be employed to improve the model's performance is early stopping. This technique can help prevent overfitting and ensure that the model is generalizing well to new data. By implementing early stopping, the model can be trained for the optimal number of epochs and achieve the best performance.

To further improve the model's performance, various techniques can be employed. Data augmentation techniques, such as flipping, rotating, and scaling, can increase the size of the training dataset and improve the model's generalization ability. Hyperparameters, such as learning rate, batch size, and number of filters, can be optimized through tuning. Transfer learning can be used to initialize the model with pre-trained weights from a similar dataset or task, improving convergence and performance. Ensemble methods can combine the predictions of multiple models trained on different subsets of data to enhance the model's robustness and performance.

Ultimately, to ensure the model's effectiveness in clinical settings, it should be validated on a diverse set of clinical data. Clinical validation is essential to verify that the model is reliable and robust in different clinical contexts, leading to better patient outcomes.