coursework_02

February 16, 2025

1 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 ###
```

1.1 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.
- 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) 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.

1.2 Dependencies

You need to install Jupyter-Lab (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]`

1.3 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.

[20]:
```
# 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.v2 as 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
```

1.4 1. Download and visualise the imaging dataset.

The dataset is curated from the brain imaging dataset in Medical Decathlon Challenge. 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

[21]:
```
# 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()
```

```
--2025-02-16 14:08:59--
https://www.dropbox.com/s/zmytk2yu284af6t/Task01_BrainTumour_2D.tar.gz
Resolving www.dropbox.com (www.dropbox.com)… 162.125.65.18,
2620:100:6025:18::a27d:4512
Connecting to www.dropbox.com (www.dropbox.com)|162.125.65.18|:443… connected.
HTTP request sent, awaiting response… 302 Found
Location: https://www.dropbox.com/scl/fi/4bf8fqcfgf3lebiv2in99/Task01_BrainTumou
r_2D.tar.gz?rlkey=ceq898g2tr3aaxjxn4xjxbob1 [following]
--2025-02-16 14:09:00--  https://www.dropbox.com/scl/fi/4bf8fqcfgf3lebiv2in99/Ta
```

```
sk01_BrainTumour_2D.tar.gz?rlkey=ceq898g2tr3aaxjxn4xjxbob1
Reusing existing connection to www.dropbox.com:443.
HTTP request sent, awaiting response… 302 Found
Location: https://uc6b491daaa439b6dd3e227282dd.dl.dropboxusercontent.com/cd/0/in
line/CkM4Zeg2RCjevHM6e8Q-MJli_0vjqPVbMUuyWK5fxLUFCl_jWM6vh5Ce5fKsldFD2OtRILAAzw-
WAvIc0K6Kx5xTjJPv0R59a537YMLLkSUx1d7YOeWwxRL8v40IgZxnLco/file# [following]
--2025-02-16 14:09:00--  https://uc6b491daaa439b6dd3e227282dd.dl.dropboxusercont
ent.com/cd/0/inline/CkM4Zeg2RCjevHM6e8Q-
MJli_0vjqPVbMUuyWK5fxLUFCl_jWM6vh5Ce5fKsldFD2OtRILAAzw-
WAvIc0K6Kx5xTjJPv0R59a537YMLLkSUx1d7YOeWwxRL8v40IgZxnLco/file
Resolving uc6b491daaa439b6dd3e227282dd.dl.dropboxusercontent.com
(uc6b491daaa439b6dd3e227282dd.dl.dropboxusercontent.com)… 162.125.65.15,
2620:100:6021:15::a27d:410f
Connecting to uc6b491daaa439b6dd3e227282dd.dl.dropboxusercontent.com
(uc6b491daaa439b6dd3e227282dd.dl.dropboxusercontent.com)|162.125.65.15|:443…
connected.
HTTP request sent, awaiting response… 302 Found
Location: /cd/0/inline2/CkNIj7PGxKqEVUlq0Z0Wwj10jA4HsVXvmMMgpgV3rZ_hfEOBLfdX5-
GrlUcTwfMg2stXJEAvadQbXfzVysfPne-x_y7Povno7nd_paRnlx8DCjZeZSX-lpCsQuIpA9Cg9H9Tpw
xezHXzvXdjAP7aCKad0tULXlRCS0cJF8h4WBWWdr2zNfGS7btcey_jomLVOpnPQ2Ntu1vIF2aYZALzpw
JDD70ciwJaDfdY72dPfQH6D9udNaieR5JiRM_n0bIf8VgPTTDWHI96Y3Bs-QW59K3DKH3nBvf26vw0-
wOmXlcemvvOv5nE8y3l_l2doFJUYZFG0oVeuEL2Rln9mwg3uM6yVgNlZuNsRgXQsCgy-lC7hQ/file
[following]
--2025-02-16 14:09:01--  https://uc6b491daaa439b6dd3e227282dd.dl.dropboxusercont
ent.com/cd/0/inline2/CkNIj7PGxKqEVUlq0Z0Wwj10jA4HsVXvmMMgpgV3rZ_hfEOBLfdX5-
GrlUcTwfMg2stXJEAvadQbXfzVysfPne-x_y7Povno7nd_paRnlx8DCjZeZSX-lpCsQuIpA9Cg9H9Tpw
xezHXzvXdjAP7aCKad0tULXlRCS0cJF8h4WBWWdr2zNfGS7btcey_jomLVOpnPQ2Ntu1vIF2aYZALzpw
JDD70ciwJaDfdY72dPfQH6D9udNaieR5JiRM_n0bIf8VgPTTDWHI96Y3Bs-QW59K3DKH3nBvf26vw0-
wOmXlcemvvOv5nE8y3l_l2doFJUYZFG0oVeuEL2Rln9mwg3uM6yVgNlZuNsRgXQsCgy-lC7hQ/file
Reusing existing connection to
uc6b491daaa439b6dd3e227282dd.dl.dropboxusercontent.com:443.
HTTP request sent, awaiting response… 200 OK
Length: 9251149 (8.8M) [application/octet-stream]
Saving to: ‘Task01_BrainTumour_2D.tar.gz.1’

Task01_BrainTumour_ 100%[===================>]   8.82M  22.5MB/s    in 0.4s

2025-02-16 14:09:01 (22.5 MB/s) - ‘Task01_BrainTumour_2D.tar.gz.1’ saved
[9251149/9251149]
```

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

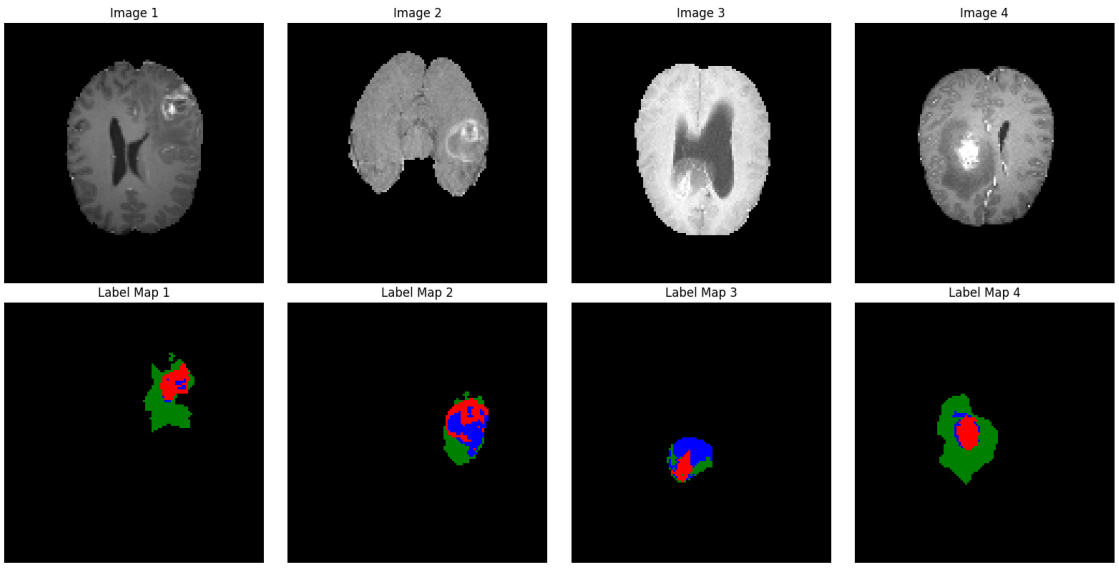
Suggested colour map for brain MR image:

```
cmap = 'gray'
```

Suggested colour map for segmentation map:

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

[22]:
```
image_dir = 'Task01_BrainTumour_2D/training_images'
label_dir = 'Task01_BrainTumour_2D/training_labels'

# Get a list of all image filenames
image_filenames = os.listdir(image_dir)
# Randomly select 4
selected_filenames = random.sample(image_filenames, 4)

# Plot figures
fig, axes = plt.subplots(2, 4, figsize=(16, 8))
for i, filename in enumerate(selected_filenames):
    image = imageio.imread(os.path.join(image_dir, filename))
    label_filename = filename.replace('_image', '_label')
    label_map = imageio.imread(os.path.join(label_dir, label_filename))

    # Display the image
    axes[0, i].imshow(image, cmap='gray')
    axes[0, i].set_title(f'Image {i+1}')
    axes[0, i].axis('off')

    # Display the label map
    cmap = colors.ListedColormap(['black', 'green', 'blue', 'red'])
    axes[1, i].imshow(label_map, cmap=cmap)
    axes[1, i].set_title(f'Label Map {i+1}')
    axes[1, i].axis('off')

plt.tight_layout()
plt.show()
```



1.6 2. Implement a dataset class.

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

[23]:
```
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 = [], []

        indices = random.sample(range(len(self.images)), batch_size)
        for idx in indices:
            image, label = self.__getitem__(idx)
            images.append(image)
            labels.append(label)

        images = np.expand_dims(np.array(images), axis=1)
        labels = np.array(labels)

        return images, labels
```

1.7 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. MICCAI, 2015.

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

[24]:
```
""" 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
        n //= 2  # 64
        self.upconv3 = nn.ConvTranspose2d(n * 2, n, kernel_size=2, stride=2)
        self.conv5 = nn.Sequential(
            nn.Conv2d(n * 2, 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.upconv2 = nn.ConvTranspose2d(n * 2, n, kernel_size=2, stride=2)
        self.conv6 = nn.Sequential(
            nn.Conv2d(n * 2, 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  # 16
        self.upconv1 = nn.ConvTranspose2d(n * 2, n, kernel_size=2, stride=2)
        self.conv7 = nn.Sequential(
            nn.Conv2d(n * 2, 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()
        )

        # Output layer
        self.final = nn.Conv2d(n, output_channel, kernel_size=1)


    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)

        # Decoder
        x = self.upconv3(x)
        x = torch.cat((x, conv3_skip), dim=1)
        x = self.conv5(x)

        x = self.upconv2(x)
```

```
        x = torch.cat((x, conv2_skip), dim=1)
        x = self.conv6(x)

        x = self.upconv1(x)
        x = torch.cat((x, conv1_skip), dim=1)
        x = self.conv7(x)

        x = self.final(x)

        return x
```

1.8 4. Train the segmentation model.

[ ]:
```
# 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)

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

# Segmentation loss
criterion = nn.CrossEntropyLoss()

# Datasets
train_set = BrainImageSet('Task01_BrainTumour_2D/training_images',␣
 ↪'Task01_BrainTumour_2D/training_labels')
test_set = BrainImageSet('Task01_BrainTumour_2D/test_images',␣
 ↪'Task01_BrainTumour_2D/test_labels')

# Train the model
# Note: when you debug the model, you may reduce the number of iterations or␣
 ↪batch size to save time.
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, e.g. dropout or batchnorm.
    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=torch.long)
    logits = model(images)

    # Perform optimisation and print out the training loss
    optimizer.zero_grad()
    loss = criterion(logits, labels)
    loss.backward()
    optimizer.step()

    # Evaluate
    if it % 100 == 0:
        model.eval()
        # Disabling gradient calculation during reference to reduce memory␣
↪consumption
        with torch.no_grad():
            # Evaluate on a batch of test images and print out the test loss
            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(device, dtype=torch.long)

            logits = model(images)
            test_loss = criterion(logits, labels)

            preds = torch.argmax(logits, dim=1)
            accuracy = (preds == labels).float().mean()

            print(f"Iteration {it}, Test Loss: {test_loss.item():.4f}, Accuracy:
↪ {accuracy.item():.4f}")

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

```
Device: cuda
Iteration 100, Test Loss: 0.5390, Accuracy: 0.9718
```

```
Iteration 200, Test Loss: 0.1772, Accuracy: 0.9866
Iteration 300, Test Loss: 0.1113, Accuracy: 0.9814
Iteration 400, Test Loss: 0.0813, Accuracy: 0.9826
Iteration 500, Test Loss: 0.0524, Accuracy: 0.9898
Iteration 600, Test Loss: 0.0620, Accuracy: 0.9807
Iteration 700, Test Loss: 0.0671, Accuracy: 0.9816
Iteration 800, Test Loss: 0.0499, Accuracy: 0.9826
Iteration 900, Test Loss: 0.0631, Accuracy: 0.9819
Iteration 1000, Test Loss: 0.0993, Accuracy: 0.9774
Iteration 1100, Test Loss: 0.0536, Accuracy: 0.9802
Iteration 1200, Test Loss: 0.0517, Accuracy: 0.9839
Iteration 1300, Test Loss: 0.0471, Accuracy: 0.9845
Iteration 1400, Test Loss: 0.0344, Accuracy: 0.9880
Iteration 1500, Test Loss: 0.0377, Accuracy: 0.9889
Iteration 1600, Test Loss: 0.0330, Accuracy: 0.9894
Iteration 1700, Test Loss: 0.0513, Accuracy: 0.9827
Iteration 1800, Test Loss: 0.0379, Accuracy: 0.9862
Iteration 1900, Test Loss: 0.0576, Accuracy: 0.9807
Iteration 2000, Test Loss: 0.0605, Accuracy: 0.9781
Iteration 2100, Test Loss: 0.0517, Accuracy: 0.9832
Iteration 2200, Test Loss: 0.0376, Accuracy: 0.9876
Iteration 2300, Test Loss: 0.0550, Accuracy: 0.9806
Iteration 2400, Test Loss: 0.0724, Accuracy: 0.9771
Iteration 2500, Test Loss: 0.0315, Accuracy: 0.9888
Iteration 2600, Test Loss: 0.0362, Accuracy: 0.9894
Iteration 2700, Test Loss: 0.0364, Accuracy: 0.9907
Iteration 2800, Test Loss: 0.0332, Accuracy: 0.9892
Iteration 2900, Test Loss: 0.0541, Accuracy: 0.9844
Iteration 3000, Test Loss: 0.0327, Accuracy: 0.9889
Iteration 3100, Test Loss: 0.0351, Accuracy: 0.9895
Iteration 3200, Test Loss: 0.0282, Accuracy: 0.9897
Iteration 3300, Test Loss: 0.0439, Accuracy: 0.9859
Iteration 3400, Test Loss: 0.0348, Accuracy: 0.9871
Iteration 3500, Test Loss: 0.0392, Accuracy: 0.9889
Iteration 3600, Test Loss: 0.0349, Accuracy: 0.9904
Iteration 3700, Test Loss: 0.0562, Accuracy: 0.9839
Iteration 3800, Test Loss: 0.0234, Accuracy: 0.9920
Iteration 3900, Test Loss: 0.0415, Accuracy: 0.9860
Iteration 4000, Test Loss: 0.0370, Accuracy: 0.9891
Iteration 4100, Test Loss: 0.0528, Accuracy: 0.9864
Iteration 4200, Test Loss: 0.0202, Accuracy: 0.9929
Iteration 4300, Test Loss: 0.0392, Accuracy: 0.9901
Iteration 4400, Test Loss: 0.0328, Accuracy: 0.9901
Iteration 4500, Test Loss: 0.0247, Accuracy: 0.9931
Iteration 4600, Test Loss: 0.0287, Accuracy: 0.9914
Iteration 4700, Test Loss: 0.0449, Accuracy: 0.9856
Iteration 4800, Test Loss: 0.0681, Accuracy: 0.9833
Iteration 4900, Test Loss: 0.0694, Accuracy: 0.9832
```

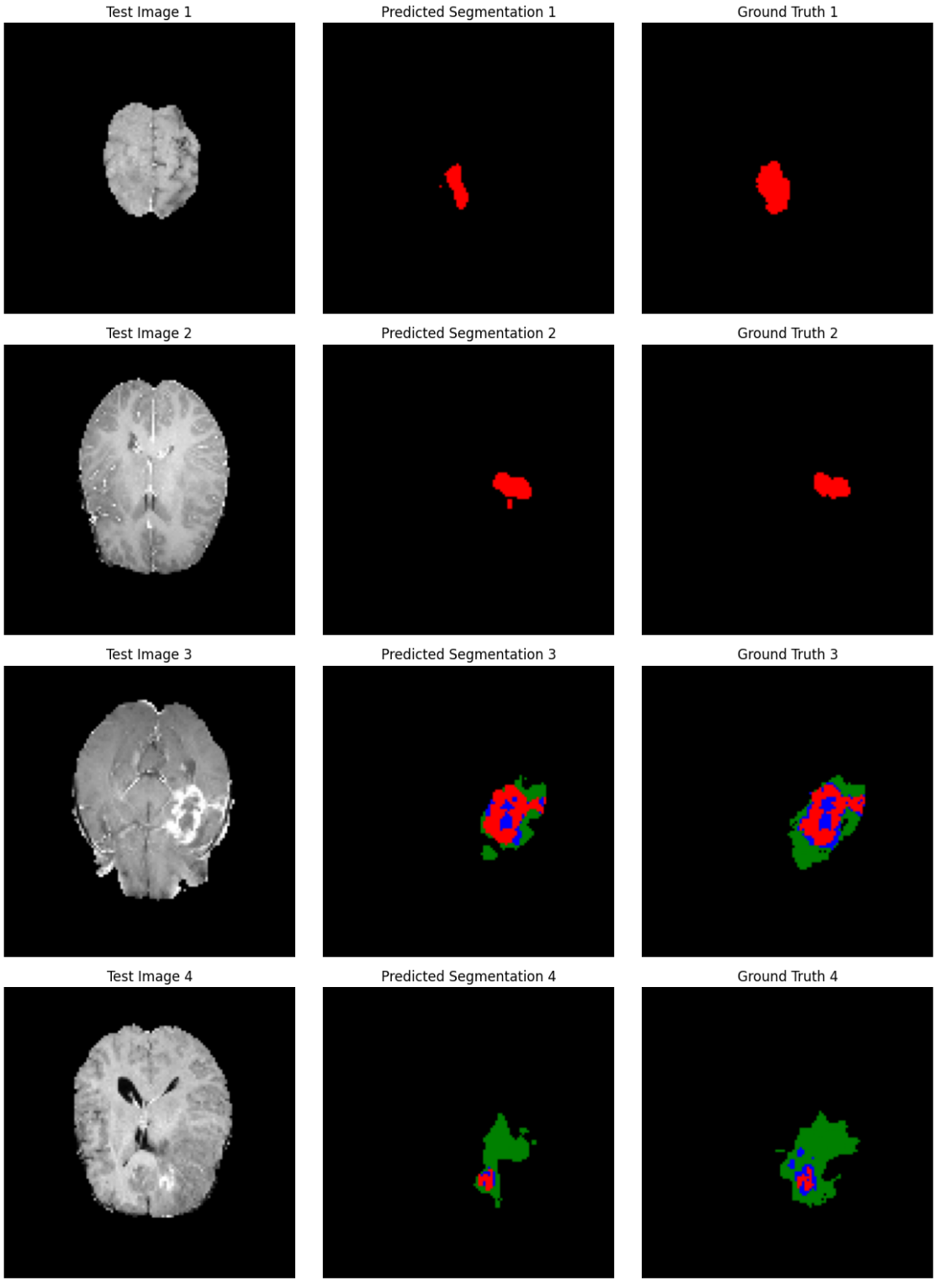
```
Iteration 5000, Test Loss: 0.0437, Accuracy: 0.9888
Iteration 5100, Test Loss: 0.0277, Accuracy: 0.9917
Iteration 5200, Test Loss: 0.0424, Accuracy: 0.9883
Iteration 5300, Test Loss: 0.0442, Accuracy: 0.9883
Iteration 5400, Test Loss: 0.0480, Accuracy: 0.9900
Iteration 5500, Test Loss: 0.0342, Accuracy: 0.9896
Iteration 5600, Test Loss: 0.0365, Accuracy: 0.9912
Iteration 5700, Test Loss: 0.0318, Accuracy: 0.9902
Iteration 5800, Test Loss: 0.0222, Accuracy: 0.9921
Iteration 5900, Test Loss: 0.0422, Accuracy: 0.9887
Iteration 6000, Test Loss: 0.0270, Accuracy: 0.9941
Iteration 6100, Test Loss: 0.0484, Accuracy: 0.9889
Iteration 6200, Test Loss: 0.0368, Accuracy: 0.9909
Iteration 6300, Test Loss: 0.0576, Accuracy: 0.9868
Iteration 6400, Test Loss: 0.0317, Accuracy: 0.9927
Iteration 6500, Test Loss: 0.0297, Accuracy: 0.9909
Iteration 6600, Test Loss: 0.0305, Accuracy: 0.9905
Iteration 6700, Test Loss: 0.0307, Accuracy: 0.9920
Iteration 6800, Test Loss: 0.0374, Accuracy: 0.9906
Iteration 6900, Test Loss: 0.0657, Accuracy: 0.9840
Iteration 7000, Test Loss: 0.0410, Accuracy: 0.9894
Iteration 7100, Test Loss: 0.0468, Accuracy: 0.9901
Iteration 7200, Test Loss: 0.0341, Accuracy: 0.9903
Iteration 7300, Test Loss: 0.0308, Accuracy: 0.9921
Iteration 7400, Test Loss: 0.0595, Accuracy: 0.9882
Iteration 7500, Test Loss: 0.0514, Accuracy: 0.9868
Iteration 7600, Test Loss: 0.0308, Accuracy: 0.9927
Iteration 7700, Test Loss: 0.0361, Accuracy: 0.9923
Iteration 7800, Test Loss: 0.0509, Accuracy: 0.9870
Iteration 7900, Test Loss: 0.0424, Accuracy: 0.9883
Iteration 8000, Test Loss: 0.0679, Accuracy: 0.9848
Iteration 8100, Test Loss: 0.0419, Accuracy: 0.9904
Iteration 8200, Test Loss: 0.0547, Accuracy: 0.9897
Iteration 8300, Test Loss: 0.0768, Accuracy: 0.9855
Iteration 8400, Test Loss: 0.0622, Accuracy: 0.9864
Iteration 8500, Test Loss: 0.0720, Accuracy: 0.9819
Iteration 8600, Test Loss: 0.0400, Accuracy: 0.9911
Iteration 8700, Test Loss: 0.0268, Accuracy: 0.9933
Iteration 8800, Test Loss: 0.0636, Accuracy: 0.9891
Iteration 8900, Test Loss: 0.0637, Accuracy: 0.9874
Iteration 9000, Test Loss: 0.0451, Accuracy: 0.9888
Iteration 9100, Test Loss: 0.0662, Accuracy: 0.9900
Iteration 9200, Test Loss: 0.0592, Accuracy: 0.9878
Iteration 9300, Test Loss: 0.0625, Accuracy: 0.9867
Iteration 9400, Test Loss: 0.0506, Accuracy: 0.9890
Iteration 9500, Test Loss: 0.0618, Accuracy: 0.9879
Iteration 9600, Test Loss: 0.0678, Accuracy: 0.9863
Iteration 9700, Test Loss: 0.0705, Accuracy: 0.9872
```

```
Iteration 9800, Test Loss: 0.0585, Accuracy: 0.9875
Iteration 9900, Test Loss: 0.0226, Accuracy: 0.9927
Iteration 10000, Test Loss: 0.0665, Accuracy: 0.9875
Training took 314.121s in total.
```

1.9 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.

[34]:
```
# Load the trained model
model_path = os.path.join(model_dir, 'model_10000.pt')
model.load_state_dict(torch.load(model_path, weights_only=True))
model.eval()

# Get a random set of 4 test images
images, labels = test_set.get_random_batch(4)
images, labels = torch.from_numpy(images), torch.from_numpy(labels)
images, labels = images.to(device, dtype=torch.float32), labels.to(device,␣
 ↪dtype=torch.long)
logits = model(images)
preds = torch.argmax(logits, dim=1).squeeze().cpu().numpy()

# Plot
fig, axes = plt.subplots(4, 3, figsize=(12, 16))

for i in range(preds.shape[0]):
    # Display the test image
    axes[i, 0].imshow(images[i, 0].cpu(), cmap='gray')
    axes[i, 0].set_title(f'Test Image {i+1}')
    axes[i, 0].axis('off')

    # Display the predicted segmentation
    cmap = colors.ListedColormap(['black', 'green', 'blue', 'red'])
    axes[i, 1].imshow(preds[i], cmap=cmap)
    axes[i, 1].set_title(f'Predicted Segmentation {i+1}')
    axes[i, 1].axis('off')

    # Display the ground truth segmentation
    axes[i, 2].imshow(labels[i].cpu(), cmap=cmap)
    axes[i, 2].set_title(f'Ground Truth {i+1}')
    axes[i, 2].axis('off')

plt.tight_layout()
plt.show()
```


Test Image 1
Predicted Segmentation 1
Ground Truth 1
Test Image 2
Predicted Segmentation 2
Ground Truth 2
Test Image 3
Predicted Segmentation 3
Ground Truth 3
Test Image 4
Predicted Segmentation 4
Ground Truth 4

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

The trained model performs quite well, reaching up to 99.27% test accuracy. It is mostly able to identify presence of anomalies, along with a rough region, aligning to the label. However, the model may sometimes mislabel the type of tumour, as well as the precise size.

In order to improve the model, it could be fine tuned using more examples of tumours, perhaps using augmented examples for better regularisation, training only the last few layers. Other architectures, such as U-net++, may also be considered for better performance. To deploy into real clinics, the prediction of the tumour/edema regions could be overlayed on the original image to help highlight these regions of interest to doctors. The model could also be further improved in clinical deployment, using medical professionals' judgement on real examples to fine tune the model.