coursework_02

February 21, 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.

[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.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

[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()
```

```
--2025-02-21 17:26:15--
https://www.dropbox.com/s/zmytk2yu284af6t/Task01_BrainTumour_2D.tar.gz
Resolving www.dropbox.com (www.dropbox.com)… 162.125.5.18,
2620:100:601d:18::a27d:512
Connecting to www.dropbox.com (www.dropbox.com)|162.125.5.18|: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]
--2025-02-21 17:26:15--  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://ucf00badbb25dbe23d64d0961dd9.dl.dropboxusercontent.com/cd/0/in
line/CkgDYOJZBYDSRDoNSk_6W31GpT0lJwWGsTdH7fKDaHXW0wjeHJRJYbbLcYb3CNG5dld60Z1VCAB
g8-sbWiSHfswX0XwzYkdp5fyxkFpCuadu7tQNSfD8V3-Rec5ikRX94h4/file# [following]
--2025-02-21 17:26:15--  https://ucf00badbb25dbe23d64d0961dd9.dl.dropboxusercont
ent.com/cd/0/inline/CkgDYOJZBYDSRDoNSk_6W31GpT0lJwWGsTdH7fKDaHXW0wjeHJRJYbbLcYb3
CNG5dld60Z1VCABg8-sbWiSHfswX0XwzYkdp5fyxkFpCuadu7tQNSfD8V3-Rec5ikRX94h4/file
Resolving ucf00badbb25dbe23d64d0961dd9.dl.dropboxusercontent.com
(ucf00badbb25dbe23d64d0961dd9.dl.dropboxusercontent.com)… 162.125.5.15,
2620:100:601d:15::a27d:50f
Connecting to ucf00badbb25dbe23d64d0961dd9.dl.dropboxusercontent.com
(ucf00badbb25dbe23d64d0961dd9.dl.dropboxusercontent.com)|162.125.5.15|:443…
connected.
HTTP request sent, awaiting response… 302 Found
Location: /cd/0/inline2/CkjYbLJce0gJOaX8mzGCtbrmfIo6hTgcFRX8Wg1WC6O7Py4Gkbe1dPlE
YY_Lm9vVedkvTuUJctJ1lSMcfbU_G_Za21RJw9NFWFhGj3EmHLZjIhDFegqQWhX_TOsBVwwMOrB8TxD2
IArGIR3ni7B5Qy8m0vhiIOv_XtXtC1WpRf7X6D2fJQtozLRTYDGPMu2EArX54BB55jfvhF9HME_oLY4a
9dzDW_i-L7opr4-kgqmwbcwVv5gIwOilnNnY-
loqQEOB8ST8hS0P8u1B6EzoOS5I2Njjt_MqTROc06ZoqsAHaAcH6FOBFV-
UZQwaen4f9xh8HeOvfDv9grPqhOiUQXx8CgnqFO8C7ZryuY-36T-tiA/file [following]
--2025-02-21 17:26:16--  https://ucf00badbb25dbe23d64d0961dd9.dl.dropboxusercont
ent.com/cd/0/inline2/CkjYbLJce0gJOaX8mzGCtbrmfIo6hTgcFRX8Wg1WC6O7Py4Gkbe1dPlEYY_
Lm9vVedkvTuUJctJ1lSMcfbU_G_Za21RJw9NFWFhGj3EmHLZjIhDFegqQWhX_TOsBVwwMOrB8TxD2IAr
GIR3ni7B5Qy8m0vhiIOv_XtXtC1WpRf7X6D2fJQtozLRTYDGPMu2EArX54BB55jfvhF9HME_oLY4a9dz
DW_i-L7opr4-kgqmwbcwVv5gIwOilnNnY-
loqQEOB8ST8hS0P8u1B6EzoOS5I2Njjt_MqTROc06ZoqsAHaAcH6FOBFV-
UZQwaen4f9xh8HeOvfDv9grPqhOiUQXx8CgnqFO8C7ZryuY-36T-tiA/file
Reusing existing connection to
ucf00badbb25dbe23d64d0961dd9.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’

Task01_BrainTumour_ 100%[===================>]   8.82M  42.2MB/s    in 0.2s

2025-02-21 17:26:17 (42.2 MB/s) - ‘Task01_BrainTumour_2D.tar.gz’ saved
[9251149/9251149]
```

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

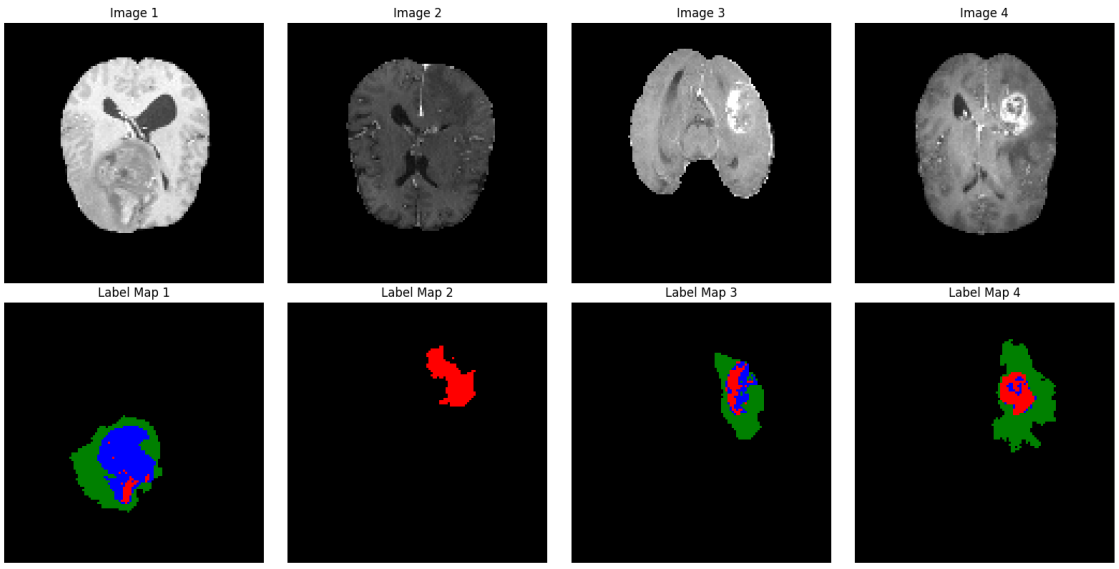
Suggested colour map for brain MR image:

```
cmap = 'gray'
```

Suggested colour map for segmentation map:

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

[3]:
```
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.

[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 = [], []

        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.

[5]:
```
""" 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.

[6]:
```
# 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}, Training Loss: {loss.item():.4f}, 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, Training Loss: 0.2729, Test Loss: 0.2683, Accuracy: 0.9815
```

```
Iteration 200, Training Loss: 0.1294, Test Loss: 0.1411, Accuracy: 0.9750
Iteration 300, Training Loss: 0.0732, Test Loss: 0.0896, Accuracy: 0.9835
Iteration 400, Training Loss: 0.0769, Test Loss: 0.0732, Accuracy: 0.9796
Iteration 500, Training Loss: 0.0480, Test Loss: 0.0657, Accuracy: 0.9866
Iteration 600, Training Loss: 0.0642, Test Loss: 0.0780, Accuracy: 0.9788
Iteration 700, Training Loss: 0.0573, Test Loss: 0.0663, Accuracy: 0.9779
Iteration 800, Training Loss: 0.0641, Test Loss: 0.0548, Accuracy: 0.9848
Iteration 900, Training Loss: 0.0318, Test Loss: 0.0644, Accuracy: 0.9791
Iteration 1000, Training Loss: 0.0298, Test Loss: 0.0421, Accuracy: 0.9886
Iteration 1100, Training Loss: 0.0506, Test Loss: 0.0444, Accuracy: 0.9881
Iteration 1200, Training Loss: 0.0498, Test Loss: 0.0740, Accuracy: 0.9754
Iteration 1300, Training Loss: 0.0282, Test Loss: 0.0672, Accuracy: 0.9808
Iteration 1400, Training Loss: 0.0336, Test Loss: 0.0269, Accuracy: 0.9917
Iteration 1500, Training Loss: 0.0435, Test Loss: 0.0531, Accuracy: 0.9827
Iteration 1600, Training Loss: 0.0365, Test Loss: 0.0338, Accuracy: 0.9892
Iteration 1700, Training Loss: 0.0292, Test Loss: 0.0472, Accuracy: 0.9872
Iteration 1800, Training Loss: 0.0243, Test Loss: 0.0515, Accuracy: 0.9839
Iteration 1900, Training Loss: 0.0237, Test Loss: 0.0533, Accuracy: 0.9816
Iteration 2000, Training Loss: 0.0358, Test Loss: 0.0420, Accuracy: 0.9878
Iteration 2100, Training Loss: 0.0313, Test Loss: 0.0310, Accuracy: 0.9896
Iteration 2200, Training Loss: 0.0238, Test Loss: 0.0418, Accuracy: 0.9862
Iteration 2300, Training Loss: 0.0277, Test Loss: 0.0361, Accuracy: 0.9892
Iteration 2400, Training Loss: 0.0378, Test Loss: 0.0325, Accuracy: 0.9881
Iteration 2500, Training Loss: 0.0191, Test Loss: 0.0576, Accuracy: 0.9832
Iteration 2600, Training Loss: 0.0257, Test Loss: 0.0347, Accuracy: 0.9865
Iteration 2700, Training Loss: 0.0117, Test Loss: 0.0492, Accuracy: 0.9861
Iteration 2800, Training Loss: 0.0157, Test Loss: 0.0474, Accuracy: 0.9846
Iteration 2900, Training Loss: 0.0233, Test Loss: 0.0596, Accuracy: 0.9843
Iteration 3000, Training Loss: 0.0246, Test Loss: 0.0470, Accuracy: 0.9870
Iteration 3100, Training Loss: 0.0194, Test Loss: 0.0365, Accuracy: 0.9904
Iteration 3200, Training Loss: 0.0262, Test Loss: 0.0287, Accuracy: 0.9911
Iteration 3300, Training Loss: 0.0215, Test Loss: 0.0547, Accuracy: 0.9848
Iteration 3400, Training Loss: 0.0192, Test Loss: 0.0413, Accuracy: 0.9863
Iteration 3500, Training Loss: 0.0249, Test Loss: 0.0457, Accuracy: 0.9861
Iteration 3600, Training Loss: 0.0208, Test Loss: 0.0313, Accuracy: 0.9896
Iteration 3700, Training Loss: 0.0160, Test Loss: 0.0540, Accuracy: 0.9872
Iteration 3800, Training Loss: 0.0157, Test Loss: 0.0418, Accuracy: 0.9891
Iteration 3900, Training Loss: 0.0162, Test Loss: 0.0376, Accuracy: 0.9918
Iteration 4000, Training Loss: 0.0156, Test Loss: 0.0265, Accuracy: 0.9927
Iteration 4100, Training Loss: 0.0152, Test Loss: 0.0320, Accuracy: 0.9907
Iteration 4200, Training Loss: 0.0125, Test Loss: 0.0232, Accuracy: 0.9931
Iteration 4300, Training Loss: 0.0132, Test Loss: 0.0457, Accuracy: 0.9862
Iteration 4400, Training Loss: 0.0162, Test Loss: 0.0347, Accuracy: 0.9892
Iteration 4500, Training Loss: 0.0144, Test Loss: 0.0302, Accuracy: 0.9926
Iteration 4600, Training Loss: 0.0241, Test Loss: 0.0466, Accuracy: 0.9894
Iteration 4700, Training Loss: 0.0155, Test Loss: 0.0327, Accuracy: 0.9891
Iteration 4800, Training Loss: 0.0114, Test Loss: 0.0461, Accuracy: 0.9872
Iteration 4900, Training Loss: 0.0118, Test Loss: 0.0333, Accuracy: 0.9908
```

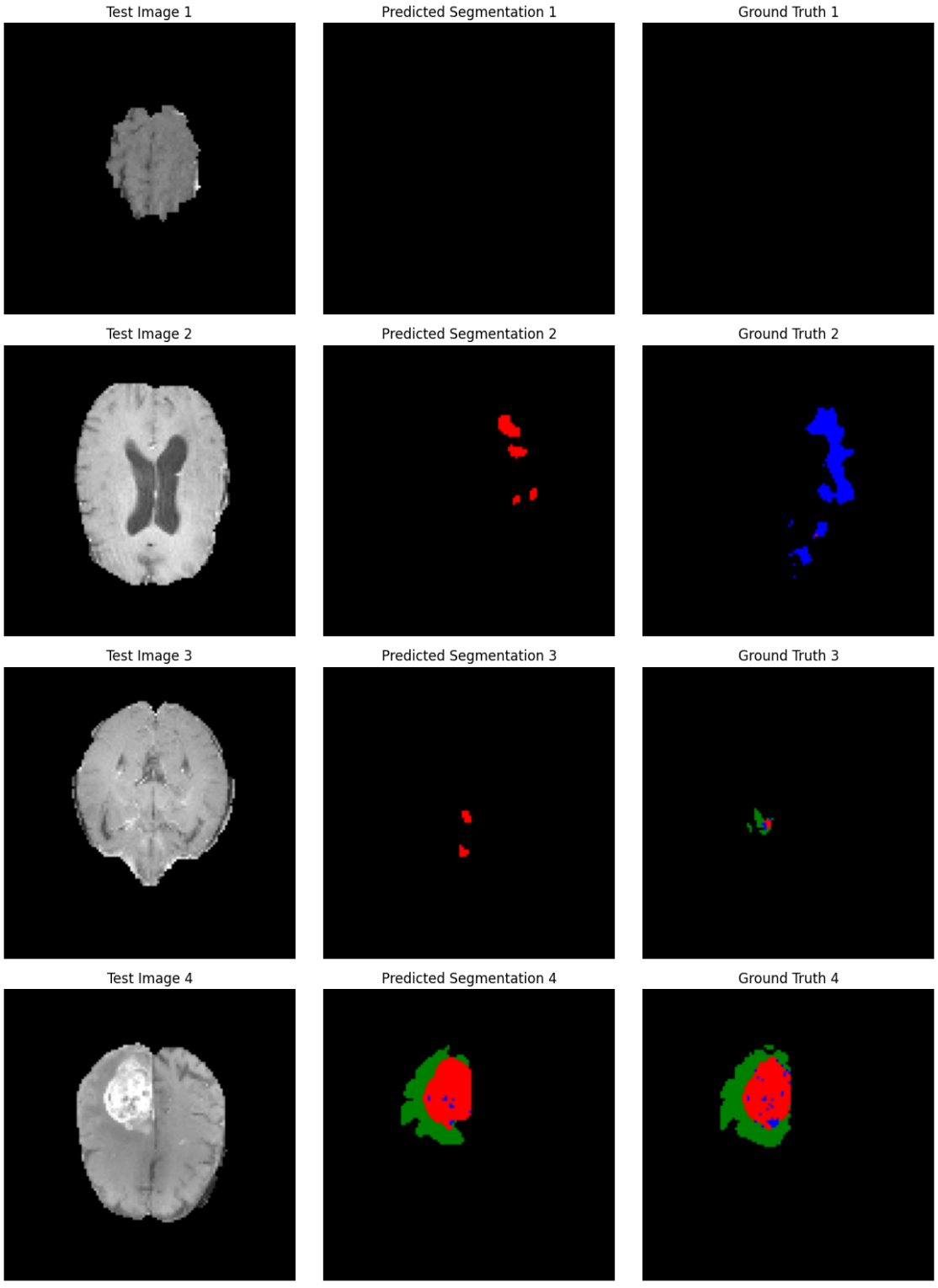
```
Iteration 5000, Training Loss: 0.0109, Test Loss: 0.0563, Accuracy: 0.9869
Iteration 5100, Training Loss: 0.0186, Test Loss: 0.0366, Accuracy: 0.9897
Iteration 5200, Training Loss: 0.0128, Test Loss: 0.0330, Accuracy: 0.9910
Iteration 5300, Training Loss: 0.0129, Test Loss: 0.0495, Accuracy: 0.9881
Iteration 5400, Training Loss: 0.0106, Test Loss: 0.0605, Accuracy: 0.9835
Iteration 5500, Training Loss: 0.0136, Test Loss: 0.0324, Accuracy: 0.9927
Iteration 5600, Training Loss: 0.0138, Test Loss: 0.0386, Accuracy: 0.9896
Iteration 5700, Training Loss: 0.0077, Test Loss: 0.0412, Accuracy: 0.9895
Iteration 5800, Training Loss: 0.0102, Test Loss: 0.0648, Accuracy: 0.9876
Iteration 5900, Training Loss: 0.0093, Test Loss: 0.0448, Accuracy: 0.9864
Iteration 6000, Training Loss: 0.0120, Test Loss: 0.0913, Accuracy: 0.9801
Iteration 6100, Training Loss: 0.0117, Test Loss: 0.0342, Accuracy: 0.9906
Iteration 6200, Training Loss: 0.0098, Test Loss: 0.0491, Accuracy: 0.9888
Iteration 6300, Training Loss: 0.0115, Test Loss: 0.0643, Accuracy: 0.9852
Iteration 6400, Training Loss: 0.0137, Test Loss: 0.0287, Accuracy: 0.9937
Iteration 6500, Training Loss: 0.0094, Test Loss: 0.0373, Accuracy: 0.9897
Iteration 6600, Training Loss: 0.0146, Test Loss: 0.0279, Accuracy: 0.9913
Iteration 6700, Training Loss: 0.0047, Test Loss: 0.0588, Accuracy: 0.9862
Iteration 6800, Training Loss: 0.0180, Test Loss: 0.0525, Accuracy: 0.9895
Iteration 6900, Training Loss: 0.0102, Test Loss: 0.0228, Accuracy: 0.9935
Iteration 7000, Training Loss: 0.0104, Test Loss: 0.0375, Accuracy: 0.9899
Iteration 7100, Training Loss: 0.0123, Test Loss: 0.0530, Accuracy: 0.9877
Iteration 7200, Training Loss: 0.0109, Test Loss: 0.0550, Accuracy: 0.9860
Iteration 7300, Training Loss: 0.0137, Test Loss: 0.0453, Accuracy: 0.9912
Iteration 7400, Training Loss: 0.0126, Test Loss: 0.0878, Accuracy: 0.9819
Iteration 7500, Training Loss: 0.0094, Test Loss: 0.0499, Accuracy: 0.9882
Iteration 7600, Training Loss: 0.0118, Test Loss: 0.0503, Accuracy: 0.9888
Iteration 7700, Training Loss: 0.0087, Test Loss: 0.0398, Accuracy: 0.9911
Iteration 7800, Training Loss: 0.0094, Test Loss: 0.0530, Accuracy: 0.9887
Iteration 7900, Training Loss: 0.0059, Test Loss: 0.0362, Accuracy: 0.9915
Iteration 8000, Training Loss: 0.0107, Test Loss: 0.0338, Accuracy: 0.9913
Iteration 8100, Training Loss: 0.0076, Test Loss: 0.0423, Accuracy: 0.9911
Iteration 8200, Training Loss: 0.0106, Test Loss: 0.0364, Accuracy: 0.9905
Iteration 8300, Training Loss: 0.0143, Test Loss: 0.0591, Accuracy: 0.9856
Iteration 8400, Training Loss: 0.0102, Test Loss: 0.0522, Accuracy: 0.9860
Iteration 8500, Training Loss: 0.0063, Test Loss: 0.0417, Accuracy: 0.9900
Iteration 8600, Training Loss: 0.0058, Test Loss: 0.0449, Accuracy: 0.9900
Iteration 8700, Training Loss: 0.0154, Test Loss: 0.0279, Accuracy: 0.9938
Iteration 8800, Training Loss: 0.0116, Test Loss: 0.0401, Accuracy: 0.9919
Iteration 8900, Training Loss: 0.0073, Test Loss: 0.0686, Accuracy: 0.9845
Iteration 9000, Training Loss: 0.0043, Test Loss: 0.0885, Accuracy: 0.9866
Iteration 9100, Training Loss: 0.0075, Test Loss: 0.0733, Accuracy: 0.9879
Iteration 9200, Training Loss: 0.0136, Test Loss: 0.0335, Accuracy: 0.9932
Iteration 9300, Training Loss: 0.0147, Test Loss: 0.0619, Accuracy: 0.9860
Iteration 9400, Training Loss: 0.0113, Test Loss: 0.0487, Accuracy: 0.9880
Iteration 9500, Training Loss: 0.0101, Test Loss: 0.0708, Accuracy: 0.9862
Iteration 9600, Training Loss: 0.0098, Test Loss: 0.0641, Accuracy: 0.9902
Iteration 9700, Training Loss: 0.0068, Test Loss: 0.0928, Accuracy: 0.9812
```

```
Iteration 9800, Training Loss: 0.0066, Test Loss: 0.0527, Accuracy: 0.9894
Iteration 9900, Training Loss: 0.0081, Test Loss: 0.0397, Accuracy: 0.9916
Iteration 10000, Training Loss: 0.0082, Test Loss: 0.0806, Accuracy: 0.9862
Training took 326.013s 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.

[7]:
```
# 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 99.32% 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. The trianing of the model is steady as the training loss continues to decrease, however, the test loss starts to oscillate, diverging from this downward trend, indicating overfitting.

In order to improve the model, it needs to combat the overfitting. Regularisation techniques such as L2 regularistion can be implemented to discourage the models reliance of critical paths, encouraging a wider distribution of smaller weights. In this scenario, this would force the model to consider more of the overall features of the images. Another technique would be to provide more diverse data samples. This could be acheived through data augmentation techniques such as small random rotations, translations, zooms, to artificially increase the training set, whilst maintaining the key features. In the case these are not sufficient enough in reducing the overfitting, early stopping using the elbow method could be used. Alternatively, a validation set could be used to constantly estimate the test performance and the model with the lowest test error could be saved. Other architectures, such as U-net++, may also be considered for better performance. Furthermore, if similar applications of this model already exist, the weights could be initialised and fine tuned, by freezing the bulk of the layers and only training the couple towards the outputs. This would increase the general capacity of the model, as well as speed up training time, allowing it to have more general knowledge, which may help in identification of medical irregularities or MRI scans.

Before deploying 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. This would need heavy testing and improvement before being used in confidence however. Although the training loss of 0.008 and test loss of 0.08 is impressive, a more important metric to focus on, and penalise, would be the recall. As this is a medical scenario with very severe consequences if a diagnosis of a tumour is missed, false negatives should be minimised through a penalty term (regularisation) in the loss function. Furthermore, other evaluation metrics such as the F1 score and recall should be analysed to ensure the balance of unnecessary false positives, as too many would make the model redundant.