

Automatic Segmentation for Lower Limb Bones & Muscles using Deep Learning

Research Compendium for ENGSCI700: Part IV Project (Project #28)

Preprocessing Training Data (DICOM MRI Scans + NIFITI Binary Segmentation Masks)

Scan Folder Name (Example: 6_AutoBindWATER_650_9B):

Select Image Data Size:

Preprocessing Inference/Test Data (Raw DICOM MRI Scans)

Scan Folder Name (Example: 6_AutoBindWATER_650_9B):

Approximate Slice Index of Pelvis (Lowest Slice Index of Any Region of Interest Being Segmented):

Run Automatic Segmentation Model (Pre-trained resnet34 U-Net)

Subject Scan File Name (Example: msk_006):

Visualisations of 2D Scans & Masks

Slice Number:

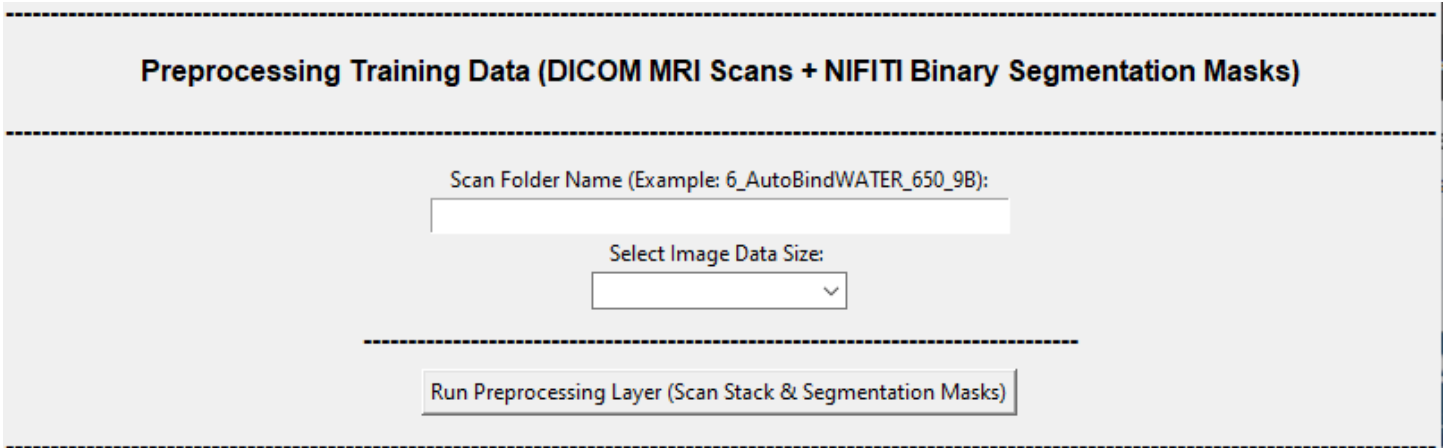
Mask File Name (Example: 4_R_tibia_5A, Example(Multi): msk_006) (Requires the preprocessed data!):

Report By: Asif Juzar Cheena, Project Partner: Pranav Rao, Project Supervisor: Dr Julie Choisne

Outline

This research compendium will go through the technical aspects of the methodology providing a solid overview of the work we did to provide some soundness to our findings. In regard to work distribution, I primarily worked on the deep learning application, and *Pranav Rao* mainly worked on data preparation as well as the data processing. We worked together on data processing, where my contribution was primarily integrating it all into the pipeline detailed in this report.

Medical Data Preprocessing Pipeline



Preprocessing Training Data (DICOM MRI Scans + NIFITI Binary Segmentation Masks)

Scan Folder Name (Example: 6_AutoBindWATER_650_9B):

Select Image Data Size:

Run Preprocessing Layer (Scan Stack & Segmentation Masks)

Figure: Training Data Processing Pipeline on GUI

This was the first major component of our deep learning segmentation pipeline and it essentially got our data in a trainable format to be fed into a deep learning architecture and actually achieve effective model learning.

Libraries

```
import numpy as np
import os
import pydicom
import SimpleITK as sitk
import pandas as pd
import trimesh
from pyntcloud import PyntCloud
from skimage.transform import resize
import sys
import matplotlib.pyplot as plt
from tqdm import tqdm
import nibabel as nib
import re
import cv2
from PIL import Image
```

Importing in the required libraries for processing the medical data preprocessing

Reading in DICOM Medical Scans and Pulling Raw Model Training Data

```
def ReadIn_MRIScans_Masks(scans_path, folders):
    print('Patient Scan Data: ', folders)
    scan_pixel_data = []
    scan_coordinate_data = []
    single_scan_pixel_data = []
    scan_coordinate_data = []
    scan_orientation_data = []
    scan_pixelspacing_data = []

    single_patient_scans_path = scans_path + '/Raw DICOM MRI Scans/{}'.format(folders)
    dicom_files = read_dicom_files(single_patient_scans_path)

    # Extracting pixel data
    for i in range(len(dicom_files)):
        normalized_data = (dicom_files[i].pixel_array)/(np.max(dicom_files[i].pixel_array))
        single_scan_pixel_data.append(normalized_data)
        print("Max pixel value in image stack is: ", np.max(normalized_data))
    scan_pixel_data.append(single_scan_pixel_data)

    training_scans = flatten_2d_array(scan_pixel_data)
    training_scans = np.array(training_scans)
    print("Max pixel value in image stack is: ", training_scans.max())

    # Coordinate Data
    single_patient_scans_path = scans_path + '/{}'.format(folders)
    for i in range(len(dicom_files)):
        scan_coordinate_data.append(dicom_files[i].ImagePositionPatient)
        scan_orientation_data.append(dicom_files[i].ImageOrientationPatient)
        scan_pixelspacing_data.append(dicom_files[i].PixelSpacing)

    coord_data = pd.DataFrame(scan_coordinate_data, columns=["x", "y", "z"])
    return training_scans
```

The function above is in charge of accessing the metadata in the DICOM MRI data. This is the main function calling helper functions like `read_dicom_files` that iteratively reads the MRI scan data folders with 1015 individual 2D DICOM scan files. The `read_dicom_files` outputs the pixel intensity data and coordinate data of each 2D slice (2D scan). 3D arrays storing the patient's scan pixel data are returned to a larger function I will detail next. This provides us with raw scan pixel data.

Reading in Binary Mask Data from 3D Slicer Software Output and Pulling Mask Data

```
def MaskCreation(basedir, filename_label):
    seg_masks_15A_tibia = sitk.ReadImage('{}\Raw NIFITI Segmentation Masks (3D Slicer Output)\{}'.format(basedir, filename_label))
    seg_masks_15A_tibia_data = sitk.GetArrayFromImage(seg_masks_15A_tibia)
    voxel_dimensions = seg_masks_15A_tibia.GetSpacing()
    voxel_dimensions = pd.DataFrame(data = voxel_dimensions)

    seg_masks_15A_tibia_data = seg_masks_15A_tibia_data[::-1, :, :]
    seg_masks_15A_tibia_data = np.where(seg_masks_15A_tibia_data != 0, 1, 0)

    seg_masks_15A_tibia_data = np.array(seg_masks_15A_tibia_data)
    indices = np.transpose(np.nonzero(seg_masks_15A_tibia_data != 0))
    if indices.size > 0:
        first_non_zero_index_2d = tuple(indices[0])
    else:
        first_non_zero_index_2d = None

    if indices.size > 0:
        last_non_zero_index_2d = tuple(indices[-1])
    else:
        last_non_zero_index_2d = None

    print("AOI Slice Start: ", first_non_zero_index_2d[0])
    print("AOI Slice End: ", last_non_zero_index_2d[0])
    print("AOI Slice Range: ", (last_non_zero_index_2d[0] - first_non_zero_index_2d[0] + 1))

    return seg_masks_15A_tibia_data, first_non_zero_index_2d[0], last_non_zero_index_2d[0]
```

This pulls the data from the binary segmentation masks created in the 3D Slicer modeling software by *Pranav* and pulls the “subsetting” slice information, and this determines where exactly in the pelvis starts and the fibula ends for a given patient. The raw data is of the patient’s full body, however we only need their lower limb data for the segmentation task as all our regions of interest lie here. So this function feeds into a larger main function similar to the previous function mentioned.

Core Data Preprocessing Function

```
def preprocessing(scans_path, filename_labels, folders, total_slices_raw_data, DataOnlyAOI, Cropping):
    train_mask_tibia_labels, training_scans, start_slices_aoi, end_slices_aoi, slice_aoi_ranges = [], [], [], [], []
    filename_labels = [filename_labels]
    print('\n')
    print('Patient Scan Data Folders Included in Run: ', folders)

    for index, filename_label in enumerate(filename_labels):
        print('\n')
        print('Segmentation Mask: ', ('{}'.format(filename_label)))

        training_scan = ReadIn_MRIScans_Masks(scans_path, folders[index])
        seg_masks_15A_tibia_data, slices_aoi_start, slices_aoi_end = MaskCreation(scans_path, filename_label)
        median_aoi_index = int(np.abs(slices_aoi_end - slices_aoi_start) / 2) + slices_aoi_start

        if(DataOnlyAOI == True):
            if (index == 0):
                train_mask_tibia_labels = seg_masks_15A_tibia_data[(slices_aoi_start):(slices_aoi_end+1)]
                training_scans = training_scan[(slices_aoi_start):(slices_aoi_end+1)]
            else:
                train_mask_tibia_labels = np.concatenate((train_mask_tibia_labels, seg_masks_15A_tibia_data[(slices_aoi_start):(slices_aoi_end+1)]), axis=0)
                training_scans = np.concatenate((training_scans, training_scan[(slices_aoi_start):(slices_aoi_end+1)]), axis=0)

        if (DataOnlyAOI == False):
            train_mask_tibia_labels.append(seg_masks_15A_tibia_data)
            training_scans.append(training_scan)

        start_slices_aoi.append(slices_aoi_start)
        end_slices_aoi.append(slices_aoi_end)

        print('\n')

    train_mask_tibia_labels = np.array(train_mask_tibia_labels)
    training_scans = np.array(training_scans)

    # Normalization and Binarization
    training_scans = training_scans.astype('float32')
    training_scans /= 255. # scale scans to [0, 1]
```

This is the main preprocessing function, the brain of the processing phase. On a patient-level we pass through a 3D array of patient scan and mask data after reading them from the previous 2 functions mentioned. The function first performs the domain-specific preprocessing step of subsetting the scan volumes to strictly lower limb scans, from the patient's waist-down.

Normalization + Binarization

```
# Normalization and Binarization
training_scans = training_scans.astype('float32')
training_scans /= 255. # scale scans to [0, 1]
train_mask_tibia_labels = np.where(train_mask_tibia_labels != 0, 1, 0)
print("Scans Normalized! [0-1]")
print("Max pixel value in image stack is: ", training_scans.max())
print("Masks Binarised! [0,1]")
print("Labels in the mask are : ", np.unique(train_mask_tibia_labels))
print("\n")
```

After that pixel scan data is min-max normalized, and the mask is binarized. Normalizing the pixel data brings it into a range of 0 to 1 for models to learn effectively. The binarizing of the mask forms a binary mask where it represents a single segmentation task of the model. The mask data would be composed of 2 classes, 0's to represent the background pixel data, and 1's to represent the pixels that belong to the segmentation task. Segmentation refers to the task of segmenting an individual bone group: tibia, femur, fibula or pelvis. Note: Sometimes the values from the 3D Slice output were not binary in nature, so the binarization step is just a safety measure.

Image Resizing and Outputting Training Data

```
if (Cropping == True):
# Resize 'training_scans'
training_scans_resized = np.empty((training_scans.shape[0], 256, 256, 1), dtype=np.float32)
for i in range(training_scans.shape[0]):
    input_image = training_scans[i, :, :, 0] # Extract the 2D image from the 4D array
    pil_image = Image.fromarray(input_image) # Convert to Pillow Image
    resized_image = pil_image.resize((256, 256), Image.BILINEAR) # Resize using Pillow
    training_scans_resized[i, :, :, 0] = np.array(resized_image) # Store the resized image in the output array

# Resize 'train_mask_tibia_labels'
train_mask_tibia_labels_resized = np.empty((train_mask_tibia_labels.shape[0], 256, 256, 1), dtype=np.uint8)
for i in range(train_mask_tibia_labels.shape[0]):
    input_image = train_mask_tibia_labels[i, :, :, 0] # Extract the 2D image from the 4D array
    pil_image = Image.fromarray(input_image) # Convert to Pillow Image
    resized_image = pil_image.resize((256, 256), Image.BILINEAR) # Resize using Pillow
    train_mask_tibia_labels_resized[i, :, :, 0] = np.array(resized_image) # Store the resized image in the output array

training_scans = training_scans_resized
train_mask_tibia_labels = train_mask_tibia_labels_resized

print('Training Scans Input Shape (Full 3D Stack): ', training_scans.shape)
print('Training Masks Input Shape (Full 3D Stack): ', train_mask_tibia_labels.shape)

return training_scans, train_mask_tibia_labels, median_aoi_index
```

The last part of the main function seen above performs image resizing to lower the dimensionality of our input. This was needed as data size 512x512 was too computationally expensive, and did not support model performance. 256x256 was a more optimal decision, to speed up the flow of patient data within the larger data pipeline, and supports better modeling computational efficiency more importantly. After resizing, the training scans and training masks are returned into the core of our data pipeline.

Core Data Pipeline (Multi-Class Segmentation Creation)

```
for segmask in segmasks:
    print(segmask)
    imgs_train, imgs_mask_train, median_aoi_index = preprocessing(scans_path, segmask, scan_data_folders, total_slices_raw_data, DataOnlyAOI, Cropping)
    orientation = (segmask.split('_'))[1]
    colab_fname = [transform_string(segmask)]
    Export2CompressedNifiti(imgs_train, scans_path, colab_fname, imgs_mask_train, orientation)

# User Input
# Multi-class mask creation
individual_mask_directory = ('{}/nnUNet Data/masks').format(scans_path)
multiclass_mask_output_dir = ('{}/nnUNet Data/multiclass_masks').format(scans_path)
scan_dir = ('{}/nnUNet Data/scans').format(scans_path)
input_scan_dir = ('{}/nnUNet Data/unprocessed_scans').format(scans_path)

TIBIA_encoding = 1
FEMUR_encoding = 2
FIBULA_encoding = 3
PELVIS_encoding = 4
mask_index = int((scan_data_folders[0].split('_'))[0])
AOIThresholding = True
FriedLanderDataset = False

print('Multi-Class Segmentation Task Data Preparation!')
CreateMasks4MulticlassMSK(input_scan_dir, scan_dir, individual_mask_directory, mask_index, TIBIA_encoding, FEMUR_encoding, FIBULA_encoding, PELVIS_encoding,

print('-'*30)
print('Completed Preprocessing Stage!')
print('-'*30)
```

Having 4 distinct deep learning models to all individually predict a single bone group is infeasible and impractical. So we expand the scope of the data and essentially concatenate all the individual binary masks into a multi-class segmentation mask that allows for a multi-class segmentation model. The **CreateMasks4MulticlassMSK** is creating the multi-class masks. It accesses all the individual binary segmentation data that was written out from the model as *.nii.gz files for a given patient. After doing that it essentially encodes each binary mask with their class value. It maps across as follows: background = 0, tibia = 1, femur = 2, fibula = 3, and pelvis = 4. Once encoded, each 2D array was summed together. In theory, with no overlapping regions the multi-class should be perfectly encoded with only the label values listed. If there are overlapping regions the encoding values would be greater than 4, as the non-zero encodings would sum when concatenating. We replace the overlap with the background so the model is not confused. Another approach would be selecting one bone group over the other, but we opted for the background encoding as it smoothens the outline of the mask.

Within **CreateMasks4MulticlassMSK** we had a function that exported out all the the preprocessed multi-class segmentation masks as *.nii.gz files into the appropriate training label data folder in the directory.

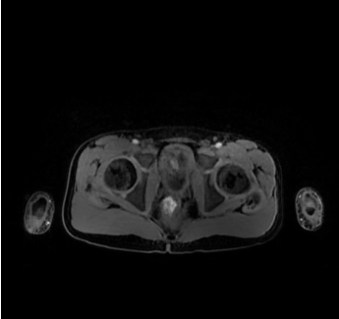
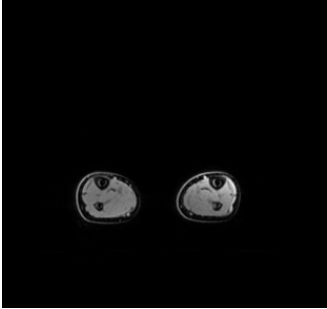
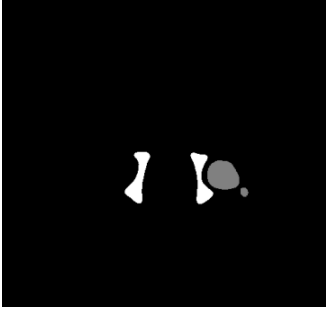
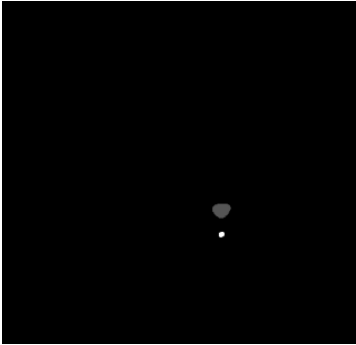
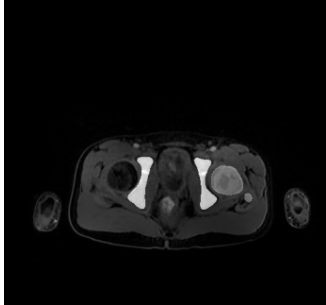
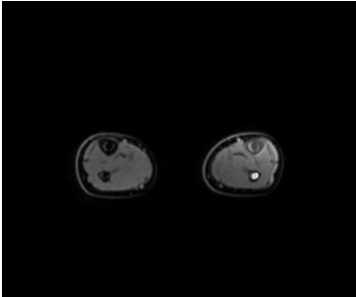
The **Export2CompressedNifiti** is the export function used in the **CreateMasks4MulticlassMSK** function and it writes the preprocessing pixel training data into the appropriate training data folder in the directory.

Preprocessed Data Visualisation (Tairawhiti Dataset)

Visualisations of 2D Scans & Masks

Slice Number:

Mask File Name (Example: 4_R_tibia_5A, Example(Multi): msk_006) (Requires the preprocessed data!):

| | | |
|------------------------------------|---|---|
| MRI Raw Scan |  |  |
| Multi-Class Mask |  |  |
| Superimposed Representation |  |  |

Deep Learning Libraries

```
import segmentation_models as sm
import keras
import tensorflow
print(keras.__version__)
print(tensorflow.__version__)

from sklearn.model_selection import train_test_split
from keras.utils import to_categorical
from keras.models import load_model
from keras.callbacks import ModelCheckpoint, EarlyStopping
from keras.metrics import MeanIoU
from keras.callbacks import Callback

import numpy as np
from matplotlib import pyplot as plt
import nibabel as nib
from tqdm import tqdm
```

Data Augmentation - Core Modelling Pipeline

```
if (training_augmentation == True):
    print('\n')
    print('-'*30)
    print('Data Augmentation Starting...')
    print('-'*30)

    img_stack_aug, msk_stack_aug = DataAugmentation(img_stack, msk_stack, num_augmentations)

    img_stack_aug = np.expand_dims(img_stack_aug, axis=-1)
    msk_stack_aug = np.expand_dims(msk_stack_aug, axis=-1)

    print('Number of Augmentation per Input: ', num_augmentations)
    print('\n')
    print('Shape of Augmented Images: ', img_stack_aug.shape)
    print('Shape of Augmented Masks: ', msk_stack_aug.shape)
    print('\n')

    if True:
        img_stack = np.concatenate((img_stack, img_stack_aug), axis=0)
        msk_stack = np.concatenate((msk_stack, msk_stack_aug), axis=0)

    print('Shape of Training Image Data: ', img_stack.shape)
    print('Shape of Training Image Masks: ', msk_stack.shape)

    print('-'*30)
    print('Completed Data Augmentation Stage!')
    print('-'*30)
    print('\n')
```

The preprocessed scan and multi-class mask data is imported into the Google Collab after uploading the *.ni.gz to Google Drive.

The above is the data augmentation main function within the data pipeline of the deep learning segmentation model.

The **DataAugmentation** function is stored in a py file and is imported. The function is fed the preprocessed training scans and mask data, and a parameter for **num_augmentation** which determines the number of augments to apply to each slice. After applying augmentation the size of the training data will scale by this factor.

Data Augmentation - Function

```
def ImageDataAugmentation(img, mask, num_augmentations):
    augmented_images = []
    augmented_masks = []

    for i in range(num_augmentations):

        # Apply a random rotation to the images
        angle = randint(-15, 15)
        M = cv2.getRotationMatrix2D((img.shape[1] / 2, img.shape[0] / 2), angle, 1)
        rotated_img = cv2.warpAffine(img, M, (img.shape[1], img.shape[0]))
        rotated_mask = cv2.warpAffine(mask, M, (mask.shape[1], mask.shape[0]))

        # Apply a random horizontal flip to the images
        if randint(0, 1):
            flipped_img = cv2.flip(rotated_img, 1)
            flipped_mask = cv2.flip(rotated_mask, 1)
        else:
            flipped_img = rotated_img
            flipped_mask = rotated_mask

        # Append the augmented images and masks to the lists
        augmented_images.append(flipped_img)
        augmented_masks.append(flipped_mask)

    # Convert the lists of augmented images and masks to NumPy arrays
    augmented_images = np.array(augmented_images)
    augmented_masks = np.array(augmented_masks)

    return augmented_images, augmented_masks

def DataAugmentation (imgs_train, imgs_mask_train, num_augmentations):
    augmented_images_train = []
    augmented_masks_train = []

    for i in tqdm(range(len(imgs_train))):
        augmented_images, augmented_masks = ImageDataAugmentation(imgs_train[i,:,:,:0], imgs_mask_train[i,:,:,:0], num_augmentations)
        augmented_images_train.append(augmented_images)
        augmented_masks_train.append(augmented_masks)

    augmented_images_train = np.array(augmented_images_train)
    augmented_masks_train = np.array(augmented_masks_train)

    augmented_images_train = flatten_array(augmented_images_train)
    augmented_masks_train = flatten_array(augmented_masks_train)

    return augmented_images_train, augmented_masks_train
```

The data augmentation function takes in 3D preprocessed scan volumes and their corresponding segmentation masks for an entire patient. It applies the following:

1. Rotation about the coronal axis (frontal axis) randomly sampled from a uniform distribution within $[-15^\circ, 15^\circ]$
2. Probabilistically flipping images about the sagittal axis from a uniform distribution within $[0, 1]$.

```
-----
Data Augmentation Starting...
-----
100%|██████████| 1656/1656 [00:07<00:00, 230.52it/s]
Number of Augmentation per Input: 1

Shape of Augmented Images: (1656, 256, 256, 1)
Shape of Augmented Masks: (1656, 256, 256, 1)

Shape of Training Image Data: (3312, 256, 256, 1)
Shape of Training Image Masks: (3312, 256, 256, 1)
-----
Completed Data Augmentation Stage!
-----
```

2D U-Net Architecture - PyTorch

```
def unet(input_shape, num_classes):
    inputs = Input(input_shape)

    # Encoder (contracting path)
    enc1, pool1 = downsample_block(inputs, 32)
    enc2, pool2 = downsample_block(pool1, 64)
    enc3, pool3 = downsample_block(pool2, 128)
    enc4, pool4 = downsample_block(pool3, 256)
    enc5, _ = downsample_block(pool4, 480)

    # Bottleneck
    bottleneck = conv_block(enc5, 480)

    # Decoder (expansive path)
    dec4 = upsample_block(bottleneck, enc4, 256)
    dec3 = upsample_block(dec4, enc3, 128)
    dec2 = upsample_block(dec3, enc2, 64)
    dec1 = upsample_block(dec2, enc1, 32)

    # Output segmentation mask
    outputs = Conv2D(num_classes, 1, activation='sigmoid')(dec1)

    model = Model(inputs, outputs)

    return model
```

```
def conv_block(inputs, filters, kernel_size=3, activation=relu):
    x = Conv2D(filters, kernel_size, padding='same')(inputs)
    x = BatchNormalization()(x)
    x = activation(x)
    x = Conv2D(filters, kernel_size, padding='same')(x)
    x = BatchNormalization()(x)
    x = activation(x)
    return x

def downsample_block(inputs, filters, kernel_size=3, activation=relu):
    x = conv_block(inputs, filters, kernel_size, activation)
    pool = MaxPooling2D()(x)
    return x, pool

def upsample_block(inputs, skip, filters, kernel_size=3, activation=relu):
    x = Conv2DTranspose(filters, kernel_size, strides=2, padding='same')(inputs)
    x = concatenate([x, skip], axis = 3)
    x = conv_block(x, filters, kernel_size, activation)
    return x
```

2D U-Net Hyper-Parameter Configuration

```
#2D U-Net Hyper-parameter Config
```

```
activation='softmax'
```

```
LR = 0.0001
```

```
optim = tensorflow.keras.optimizers.Adam(LR)
```

```
dice_loss = sm.losses.DiceLoss()
```

```
num_classes = 5
```

```
input_shape = (256, 256, 1)
```

```
early_stopping = EarlyStopping(monitor='val_loss', patience=20, restore_best_weights=True)
```

```
checkpoint = ModelCheckpoint('/content/drive/MyDrive/Colab Notebooks/2dunet_best.hdf5', monitor='val_f1-score', save_best_only=True, mode='max')
```

```
model = unet_model(input_shape, num_classes)
```

```
model.compile(optim, total_loss, metrics=metrics)
```

```
history=model.fit(X_train,  
                 y_train_cat,  
                 batch_size=30,  
                 epochs=50,  
                 verbose=1,  
                 validation_data=(X_test, y_test_cat),  
                 callbacks=[checkpoint])  
  
model.save('/content/drive/MyDrive/Colab Notebooks/2dunet_final.keras')
```

nnU-Net - Preprocessing Plan

The nnU-Net uses the same import function as stated for the 2D-UNet

```
Preprocessing

[ ] run_Proprocessing = True
    if (run_Proprocessing == True):
        !nnUNetv2_plan_and_preprocess -d 001 --verify_dataset_integrity

Fingerprint extraction...
Dataset001_Tibia
Using <class 'nnunetv2.imageio.simpleitk_reader_writer.SimpleITKIO'> as reader/writer

#####
verify_dataset_integrity Done.
If you didn't see any error messages then your dataset is most likely OK!
#####
```

nnU-Net - 2D U-Net Configuration

```
Training

▶ # 3d_fullres, 3d_lowres , 2d, 3d_cascade_fullres
  run_ModelTraining = True
  if (run_ModelTraining == True):
      !nnUNetv2_train 001 2d 4
```

nnU-Net - 3D U-Net Configuration

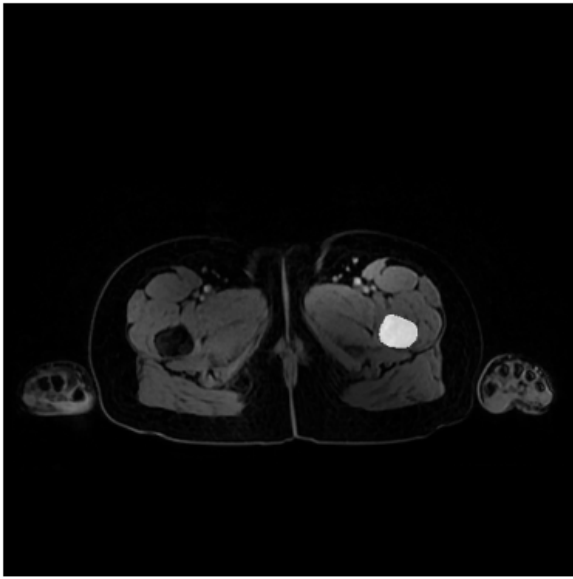
```
Training

▶ # 3d_fullres, 3d_lowres , 2d, 3d_cascade_fullres
  run_ModelTraining = True
  if (run_ModelTraining == True):
      !nnUNetv2_train 001 3d_fullres 4
```

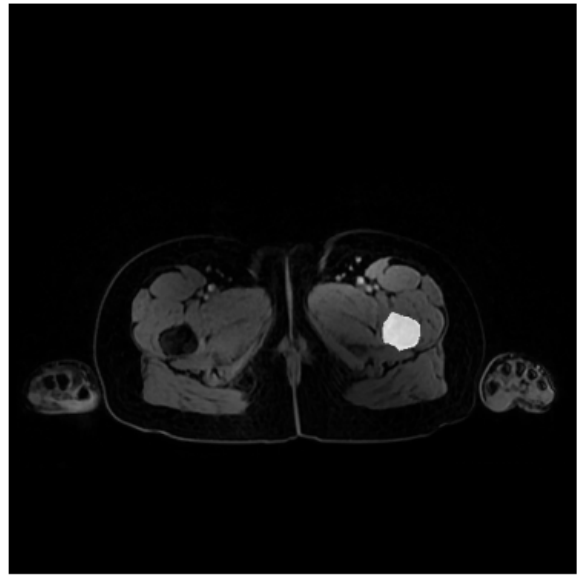
The model architecture summary output, and hyper-parameters are stored in the json files created in the preprocessing step.

nnU-Net Slice Predictions

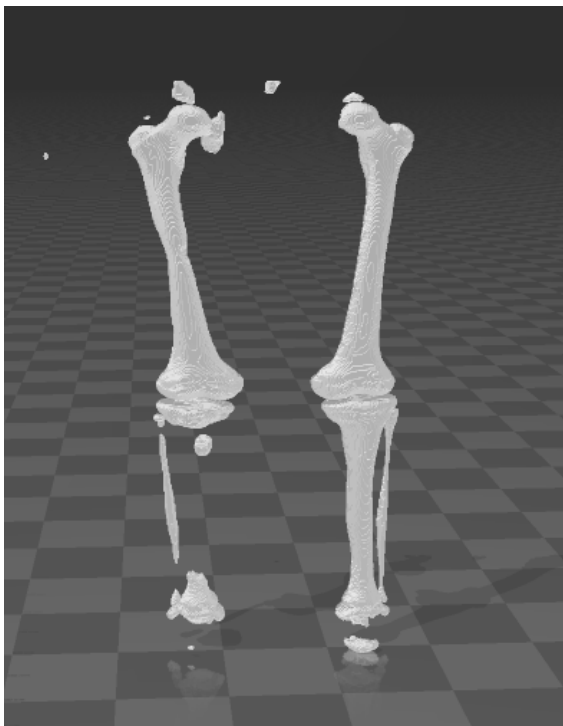
Manual Segmentation Result on Slice 50



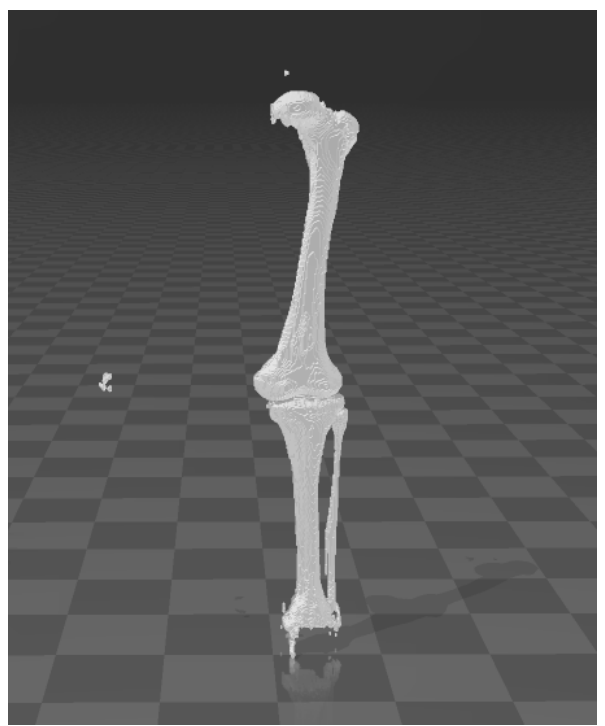
Automatic Segmentation Result on Slice 50



nnU-Net (3D U-Net) - 3D Raw Segmentation Output



nnU-Net (2D U-Net) - 3D Raw Segmentation Output



Transfer Learning Model (Resnet34 U-Net)

The transfer learning model utilizes the same scan and mask import and data augmentation functions as stated earlier.

Resnet34 U-Net Hyper-Parameter Configuration

```
activation='softmax'

initial_lr = 0.0001
optim = tensorflow.keras.optimizers.Adam(learning_rate=initial_lr)
def lr_schedule(epoch):
    return initial_lr - (0.000001 * epoch)
lr_scheduler = tensorflow.keras.callbacks.LearningRateScheduler(lr_schedule)

dice_loss = sm.losses.DiceLoss()
focal_loss = sm.losses.CategoricalFocalLoss()
total_loss = dice_loss + focal_loss

# total_loss = sm.losses.binary_focal_dice_loss # or sm.losses.categorical_focal_dice_loss

metrics = [
    sm.metrics.IOUScore(threshold=0.5),
    sm.metrics.FScore(threshold=0.5),
    dice_score
]
```

- LR Schedule (Decay of 1e-6) with LR = 1e-4
- Softmax on output layer
- Custom Loss Function of Dice Loss & Categorical Focal Loss
- After each epoch report the IoU, F1-Score, and approximate DSC

Resnet34 U-Net Model

```
# Model
BACKBONE = 'resnet34'
# BACKBONE = 'resnet50'
model = sm.Unet(BACKBONE, encoder_weights = 'imagenet', classes=n_classes, activation=activation)

preprocess_input = sm.get_preprocessing(BACKBONE)
X_train_processed = preprocess_input(X_train)
X_test_processed = preprocess_input(X_test)

# imagenet pre-trained weights
# Freeze encoder weights
model = sm.Unet(BACKBONE, encoder_weights = 'imagenet', classes=n_classes, activation=activation, encoder_freeze=True)

# Fine-tune encoder and decoder weights
# model = sm.Unet(BACKBONE, encoder_weights = 'imagenet', classes=n_classes, activation=activation)

save_model_name = '/content/drive/MyDrive/Colab Notebooks/resnet50_backbone_40_epochs_dicefocal_256_5P_24batch_maxF1_pelvis_aug'
dice_score_callback = DiceScoreCallback(validation_data=(X_test_processed, y_test_cat))
checkpoint = ModelCheckpoint(save_model_name, monitor='val_f1-score', save_best_only=True, mode='max')
early_stopping = EarlyStopping(monitor='val_loss', patience=20, restore_best_weights=True)

model.compile(optim, total_loss, metrics=metrics)
# model.compile(optim, dice_loss, metrics=metrics)

print(BACKBONE)
print(model.summary())
```

- Load in the resnet34 backbone architecture using the segmentation-models module
- Applying the required preprocessing steps for the backbone model, this was included as a safety measure as the data should have the correct dimensionality for any of the backbone models used in the experimentation.
- Constructing the model with the backbone architecture. This involved defining the set of pretrained weights for the encoder (imagenet dataset), the number of classes being the number of segmentation groups plus the background, the activation function for the output layer (softmax for multiple classes), and finally specifying whether or not we want to freeze the encoder weights which essentially freezes all the imagenet pre-trained weights of the resnet34 model. When fine-tuning with this set to true, the model would only further optimize the decoder weights. When set to false, the encoder and decoder weights are fine-tuned.
- Early stopping so if the validation loss demonstrates convergence for 20 epochs or more the training will end

Resnet34 U-Net Segmentation Predictions

```
#Set compile=False as we are not loading it for training, only for prediction.
model_fname = '/content/drive/MyDrive/Colab Notebooks/res34_backbone_20_epochs_dicefocal_256_4P_12batch_maxF1_pelvis_aug_final.keras'
model = load_model(model_fname, compile=False)
BACKBONE = 'resnet50'
pred_name = 'msk_002'

n_classes = 5

print(('Fined-Tuned Model: {}'.format((model_fname.split('/')[-1])))
print('Prediction Scan Stack: ', pred_name)
print('Number of Segmentation Classes: ', n_classes)
print('\n')

img = nib.load("/content/drive/MyDrive/Colab Notebooks/nnUNet_raw/Dataset001_Tibia/imagesTr/{}_0000.nii.gz".format(pred_name))
# img = nib.load("/content/drive/MyDrive/Colab Notebooks/nnUNet_raw/Dataset001_Tibia/imagesTs/{}_0000.nii.gz".format(pred_name))
img_data = img.get_fdata()
X_test = np.repeat(img_data, 3, axis=3)

msk = nib.load("/content/drive/MyDrive/Colab Notebooks/nnUNet_raw/Dataset001_Tibia/labelsTr/{}.nii.gz".format(pred_name))
# msk = nib.load("/content/drive/MyDrive/Colab Notebooks/nnUNet_raw/Dataset001_Tibia/imagesTs/{}.nii.gz".format(pred_name))
y_test = msk.get_fdata()

print("Test Images Shape: ", X_test.shape)
print("Test Masks Shape: ", y_test.shape)
print("Test Labels: ", np.unique(y_test))
print('\n')

# Model
# preprocess input
preprocess_input = sm.get_preprocessing(BACKBONE)
X_test_processed = preprocess_input(X_test)
test_masks_cat = to_categorical(y_test, num_classes=n_classes)
y_test_cat = test_masks_cat.reshape((y_test.shape[0], y_test.shape[1], y_test.shape[2], n_classes))

# Prediction
y_pred=model.predict(X_test_processed)
y_pred_argmax=np.argmax(y_pred, axis=3)
y_pred_argmax = np.expand_dims(y_pred_argmax, axis = -1)
print('Pred Mask Shape: ', y_pred_argmax.shape)
print("Pred Mask Labels: ", np.unique(y_pred_argmax))
print('\n')
```

- The above loads in the weights of the trained model. It also imports unseen patient's MRI and mask data. This data is preprocessed and formatted, then run through the model for prediction, producing an output/predicted segmentation.

```

tibia_seg_data_pred = np.where(y_pred_argmax != 1, 0, 1)
femur_seg_data_pred = np.where(y_pred_argmax != 2, 0, 1)
fibula_seg_data_pred = np.where(y_pred_argmax != 3, 0, 1)
pelvis_seg_data_pred = np.where(y_pred_argmax != 4, 0, 1)

tibia_seg_data_gt = np.where(y_test != 1, 0, 1)
femur_seg_data_gt = np.where(y_test != 2, 0, 1)
fibula_seg_data_gt = np.where(y_test != 3, 0, 1)
pelvis_seg_data_gt = np.where(y_test != 4, 0, 1)

dice_score_tibia = dice_coefficient(tibia_seg_data_gt, tibia_seg_data_pred)
dice_score_femur = dice_coefficient(femur_seg_data_gt, femur_seg_data_pred)
dice_score_fibula = dice_coefficient(fibula_seg_data_gt, fibula_seg_data_pred)
dice_score_pelvis = dice_coefficient(pelvis_seg_data_gt, pelvis_seg_data_pred)

TIBIA_VError = volume_error(tibia_seg_data_gt, tibia_seg_data_pred)
FEMUR_VError = volume_error(femur_seg_data_gt, femur_seg_data_pred)
FIBULA_VError = volume_error(fibula_seg_data_gt, fibula_seg_data_pred)
PELVIS_VError = volume_error(pelvis_seg_data_gt, pelvis_seg_data_pred)

print('Dice Score - Tibia:', dice_score_tibia)
print('Dice Score - Femur:', dice_score_femur)
print('Dice Score - Fibula:', dice_score_fibula)
print('Dice Score - Pelvis:', dice_score_pelvis)
print('\n')

print('VError - Tibia:', TIBIA_VError)
print('VError - Femur:', FEMUR_VError)
print('VError - Fibula:', FIBULA_VError)
print('VError - Pelvis:', PELVIS_VError)

```

- Each output segmentation mask contains segmentation data on all the tasks, so they are separated out so we can assess the performance of the model on both a multi-class and individual task basis. Each prediction segmentation mask and its corresponding ground truth mask are fed into the performance evaluation, where the metric is returned, and this encapsulates the procedure behind a single fold evaluation of a model.

Model Evaluation Metrics

```
def dice_coefficient(true_array, pred_array):
    true_array = np.asarray(true_array).astype(bool)
    pred_array = np.asarray(pred_array).astype(bool)

    intersection = np.logical_and(true_array, pred_array)
    dice = 2.0 * intersection.sum() / (true_array.sum() + pred_array.sum())
    return round(dice, 2)

def volume_error(y_true, y_pred):
    # mm3
    Voxel_Volume = 2.63671875

    N_true = np.count_nonzero(y_true)
    N_pred = np.count_nonzero(y_pred)

    volume_error = np.abs(N_true - N_pred) * Voxel_Volume

    return round(volume_error * 0.001, 2)
```

The functions calculate the Dice Similarity Score (DSC) and the Volume Error (VError) that we use to evaluate and benchmark model performance/segmentation accuracy.

3D Visualisation of Segmentation Results

```
%pip install SimpleITK
import SimpleITK as sitk
import numpy as np
from skimage.measure import marching_cubes
%pip install trimesh
import trimesh

predfname = 'msk_002'
model_used = 'resnet34'

def Export3DStructure(segmentation_data, predfname, class_msk, model_used):
    # Generate a surface mesh using marching cubes
    vertices, faces, normals, _ = marching_cubes(segmentation_data, level=0)

    # Create a Trimesh object
    mesh = trimesh.Trimesh(vertices=vertices, faces=faces, vertex_normals=normals)

    # Save the mesh as a PLY file
    ply_path = ('/content/drive/MyDrive/Colab Notebooks/nnUNet_results/Dataset001_Tibia/predTs/Transfer Learning
    mesh.export(ply_path)

segmentation_data_all = y_pred_argmax
segmentation_data_all = segmentation_data_all[:, :, :, 0]

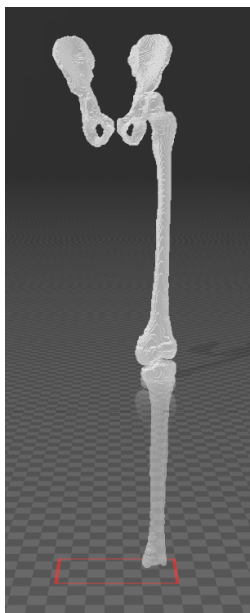
tibia_seg_data = np.where(segmentation_data_all != 1, 0, 1)
femur_seg_data = np.where(segmentation_data_all != 2, 0, 2)
fibula_seg_data = np.where(segmentation_data_all != 3, 0, 3)
pelvis_seg_data = np.where(segmentation_data_all != 4, 0, 4)

segmentation_data = [segmentation_data_all, tibia_seg_data, femur_seg_data, fibula_seg_data, pelvis_seg_data]
class_msk = ['ALL', 'TIBIA', 'FEMUR', 'FIBULA']

colors = [(255, 0, 0, 255), # Red
          (0, 255, 0, 255), # Green
          (0, 0, 255, 255), # Blue
          (255, 255, 0, 255)] # Yellow

for i in range(len(segmentation_data)):
    Export3DStructure(segmentation_data[i], predfname, class_msk[i], model_used)
```

- The function above produces PLY files containing the 3D musculoskeletal structures for each segmentation task.



Post-Processing

```
def denoising_algo(segmented_volume, min_region_size_ratio=0.01):
    depth, height, width = segmented_volume.shape

    post_processed_volume = np.zeros_like(segmented_volume)

    for z in range(depth):
        segmented_image = segmented_volume[z, :, :]

        # Hole Filling
        inverted_image = 1 - segmented_image
        labeled_components, num_components = ndimage.label(inverted_image)
        total_aoi_pixels = np.sum(segmented_image)
        min_region_size = int(min_region_size_ratio * total_aoi_pixels)

        for component in range(1, num_components + 1):
            component_size = np.sum(labeled_components == component)
            if component_size < min_region_size:
                segmented_image[labeled_components == component] = 1

        # Remove small noise using adaptive thresholding (small artifact removal)
        labeled_components, num_components = ndimage.label(segmented_image)
        min_noise_size = int(min_region_size_ratio * total_aoi_pixels)

        for component in range(1, num_components + 1):
            component_size = np.sum(labeled_components == component)
            if component_size < min_noise_size:
                segmented_image[labeled_components == component] = 0

        post_processed_volume[z, :, :] = segmented_image

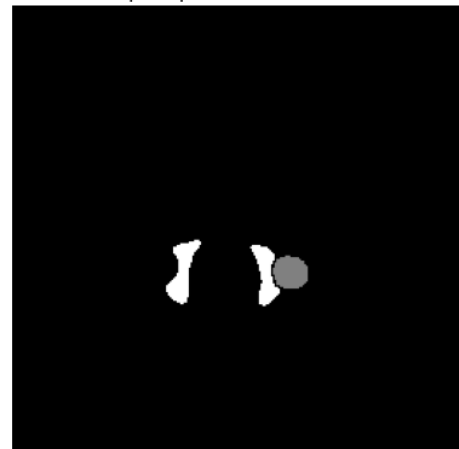
    return post_processed_volume
```

- The function above is the denoising algorithm developed for the post-processing of segmentation results
- The denoising algorithm performs both hole filling and small noise/small artifact removal, and this is done through leveraging adaptive thresholding.
- Small Artifact Removal - On a slice level we look at the pixel distribution among the classes, and then determine a minimum noise threshold. We look at given class pixels and their adjacent pixel neighbors to understand the density. If that density falls under a certain threshold the pixels of that region are considered noise and are removed.
- Hole Filling - This uses similar logic but searches for pixel density regions that are “inconsistent”. We firstly invert the encodings, get the number of pixels for each class, if this number falls under a region size threshold the reversed encodings are defined as a hole for that given class and its replaced with the class encoding.

pred mask



post-processed mask



Lower Limb Musculoskeletal Automatic Segmentation Tool

```
D:/MRI - Tairawhiti (User POV)\nnUNet Data exists.
D:/MRI - Tairawhiti (User POV)\Raw NIFITI Segmentation Masks (3D Slicer Output) exists.
D:/MRI - Tairawhiti (User POV)\Pre-Trained Models (Google Colab) exists.
D:/MRI - Tairawhiti (User POV)\Model Code exists.
D:/MRI - Tairawhiti (User POV)\Raw DICOM MRI Scans exists.
D:/MRI - Tairawhiti (User POV)\Patient Segmentation Tasks (Google Colab) exists.
D:/MRI - Tairawhiti (User POV)\nnUNet Data\masks exists.
D:/MRI - Tairawhiti (User POV)\nnUNet Data\multiclass_masks exists.
D:/MRI - Tairawhiti (User POV)\nnUNet Data\scans exists.
D:/MRI - Tairawhiti (User POV)\nnUNet Data.masks\FEMUR exists.
D:/MRI - Tairawhiti (User POV)\nnUNet Data.masks\FIBULA exists.
D:/MRI - Tairawhiti (User POV)\nnUNet Data.masks\TIBIA exists.
D:/MRI - Tairawhiti (User POV)\nnUNet Data.masks\PELVIS exists.
```

| | | | |
|--|---------------------|--------------------|------|
| Model Code | 14/09/2023 11:48 PM | File folder | |
| nnUNet Data | 8/09/2023 2:20 PM | File folder | |
| Patient Segmentation Tasks (Google Cola... | 1/09/2023 6:17 PM | File folder | |
| Pre-Trained Models (Google Colab) | 11/10/2023 9:12 AM | File folder | |
| Raw DICOM MRI Scans | 17/09/2023 11:01 PM | File folder | |
| Raw NIFITI Segmentation Masks (3D Slice... | 27/09/2023 12:01 AM | File folder | |
| .gitattributes | 31/08/2023 2:44 PM | Text Document | 1 KB |
| AutoSegTool.py | 11/10/2023 9:07 AM | Python Source File | 8 KB |

- Once the tool is launched it automatically creates all the data directories required to run the pipeline
 - Model Code - Holds all the pipeline backend code plus scripts that operate independently outside of the pipeline that yet require integration.
 - nnUNet Data - Holds all scan and multi-class/binary preprocessed outputs (**Note:** Despite its naming it holds ALL preprocessed data that is fed into all deep learning models)
 - Patient Segmentation Tasks (Google Colab) - Holds all the preprocessed test instances that do not have any ground truth segmentations
 - Pre-Trained Models (Google Colab) - This holds all the pre-trained deep learning models as *.keras and *.h5 files containing their weights
 - Raw DICOM MRI Scans - This holds all the raw patient data as 2D DICOM files
 - Raw NIFITI Segmentation Masks (3D Slicer Output) - This contains all the binarized segmentation masks

Base Directory (Example: D:/MRI - Tairawhiti):

- Define the base directory, essentially where the folders stated above lie

Preprocessing Training Data (DICOM MRI Scans + NIFITI Binary Segmentation Masks)

Scan Folder Name (Example: 6_AutoBindWATER_650_9B):

Select Image Data Size:

Run Preprocessing Layer (Scan Stack & Segmentation Masks)

- This part of the tool runs the preprocessing data pipeline
 - Scan Folder Name - The name of the patient scan folder within the Raw DICOM MRI Scans folder
 - Select Image Data Size - Provides a dropbox that allows the resizing of the preprocessed data as either 512x512 or 256x256.
 - Each patient folder has a specific de-identified code and that allows us to clearly define on a patient level what data belongs to who. When entering the scan folder name it's important to follow the structure, mainly the starting index and the final code (6, 9B). As long as the binarized mask data imported in from 3D slicer follows this structure the pipeline picks it up from its given folder and runs the multi-class segmentation creation.
 - Output: Preprocessed Scan Data & Multiclass Segmentation Masks

Preprocessing Inference/Test Data (Raw DICOM MRI Scans)

Scan Folder Name (Example: 6_AutoBindWATER_650_9B):

Approximate Slice Index of Pelvis (Lowest Slice Index of Any Region of Interest Being Segmented):

Generate Preprocessed Inference/Prediction (Scan Stack)

- Preprocessing inference data runs a special variation. This is not for model training and for preprocessing data that does not have any binarized masks/manual segmentation data available. It is for running predictions/inference.
- Scan Folder Name - Similar to above!
- Approximate Slice Index of Pelvis - This information is required from the user, and it can be an estimate (400-600), and it simply helps the model best subset the lower limbs of the patient and it is required if a full body scan is inputted into the model.

Run Automatic Segmentation Model (Pre-trained resnet34 U-Net)

Subject Scan File Name (Example: msk_006):

Generate Segmentation Output

- This runs the deep learning segmentation model
- Subject Scan File Name - This is the only required input, and it is a preprocessed compressed NIFTI (*.nii.gz) containing preprocessed scan data. Notice the naming convention. It correlates to the raw scan data where that leading index is used to define the preprocessed data.

Visualisations of 2D Scans & Masks

Slice Number:

Mask File Name (Example: 4_R_tibia_5A, Example(Multi): msk_006) (Requires the preprocessed data!):

- This part of the tool was very useful in regard to developing our model, visualizing segmentation output results on a 2D slice level, and also data validation to ensure the scan and mask data have the correct spatial overlap.
- Slice Number - The index of the slice you want to visualize
- Mask File Name - The function requires the mask file name and with that it pulls the scan data from the preprocessed scan folder. Notice the naming convention required once again.
- When RUN it produces 3 visualizations: MRI scan slice, multi-class segmentation mask, and a superimposed representation of them.

Base Directory (Example: D:/MRI - Tairawhiti):

Preprocessing Training Data (DICOM MRI Scans + NIFITI Binary Segmentation Masks)

Scan Folder Name (Example: 6_AutoBindWATER_650_9B):

Select Image Data Size:

Run Preprocessing Layer (Scan Stack & Segmentation Masks)

Preprocessing Inference/Test Data (Raw DICOM MRI Scans)

Scan Folder Name (Example: 6_AutoBindWATER_650_9B):

Approximate Slice Index of Pelvis (Lowest Slice Index of Any Region of Interest Being Segmented):

Generate Preprocessed Inference/Prediction (Scan Stack)

Run Automatic Segmentation Model (Pre-trained resnet34 U-Net)

Subject Scan File Name (Example: msk_006):

Generate Segmentation Output

Visualisations of 2D Scans & Masks

Slice Number:

Mask File Name (Example: 4_R_tibia_5A, Example(Multi): msk_006) (Requires the preprocessed data!):

Experimental Result Documentation

- Our experimental results were documented on Confluence and this documentation included:
 - Information on the model configuration: deep learning architecture, number of epochs, loss function, number of training patients, batch size, and augmentation factor.
 - Dice Scores across segmentation tasks for given fold
 - 2D visualization of a random slice across the predicted mask
 - Training/validation loss curves
 - Training/validation IoU curves

```
Finet-Tuned Model: res34_backbone_20_epochs_dicefocal_256_4P_12batch_maxF1_pelvis_aug_best.hdf5
Prediction Scan Stack: msk_021
Number of Segmentation Classes: 5

Test Images Shape: (501, 256, 256, 3)
Test Masks Shape: (501, 256, 256, 1)
Test Labels: [0. 1. 2. 3. 4.]

16/16 [=====] - 46s 3s/step
Pred Mask Shape: (501, 256, 256, 1)
Pred Mask Labels: [0 1 2 3 4]

<ipython-input-2-5e4f7e009a2a>:45: FutureWarning: Image data has type int64, which may cause incompatibilities with other tools. This will error in NiBabel 5.0. This warning can
combined_img = nib.Nifti1Image(combined_mask, msk.affine)
Exported Prediction Segmentation: /content/drive/MyDrive/Colab Notebooks/nnUNet_results/Dataset001_Tibia/Transfer Learning/msk_021_pred.nii.gz

Dice Score - Tibia: 0.93
Dice Score - Femur: 0.93
Dice Score - Fibula: 0.75
Dice Score - Pelvis: 0.88
```

