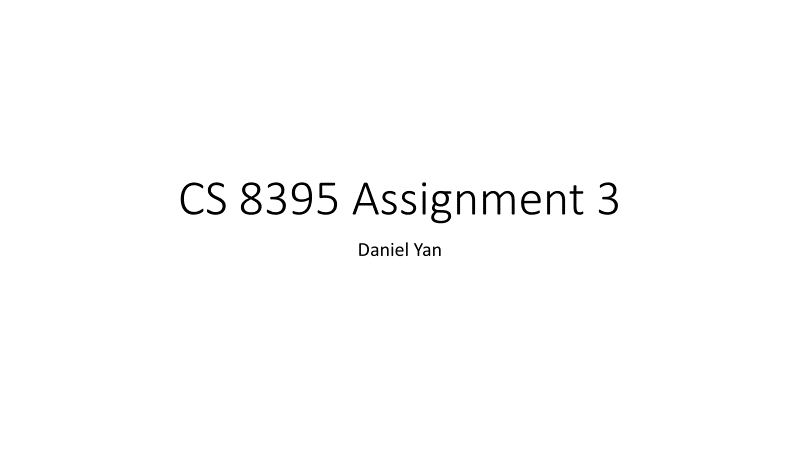
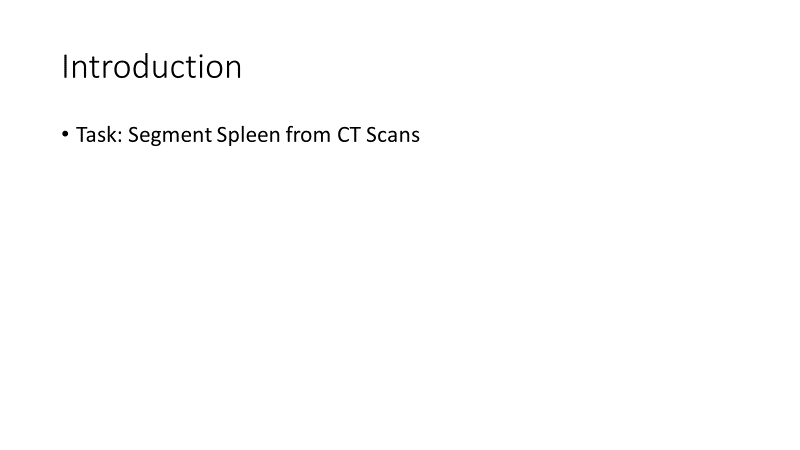
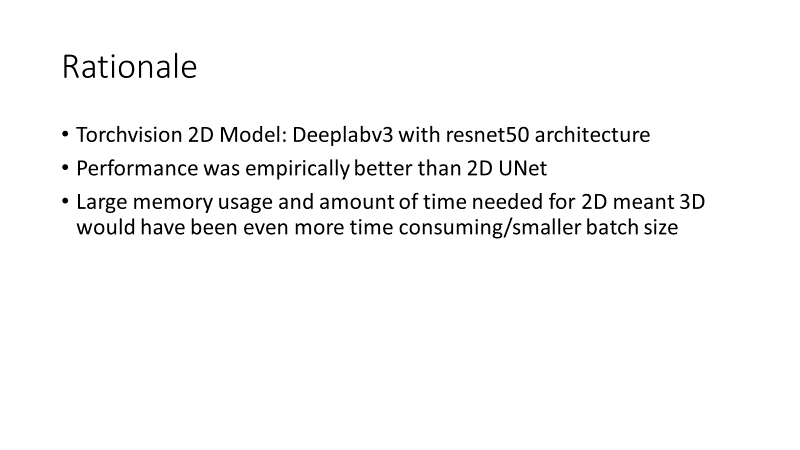
Slide 1



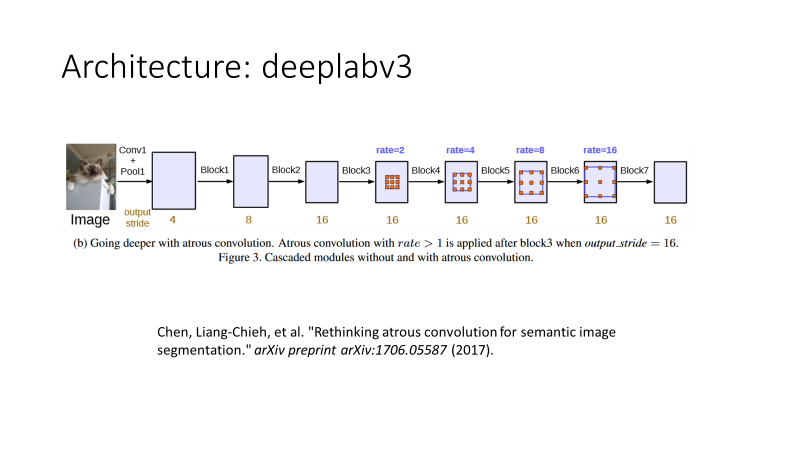
Slide 2



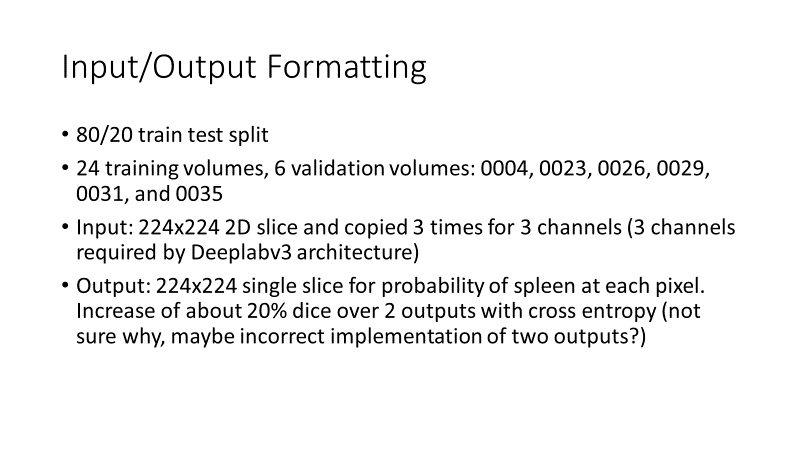
Slide 3



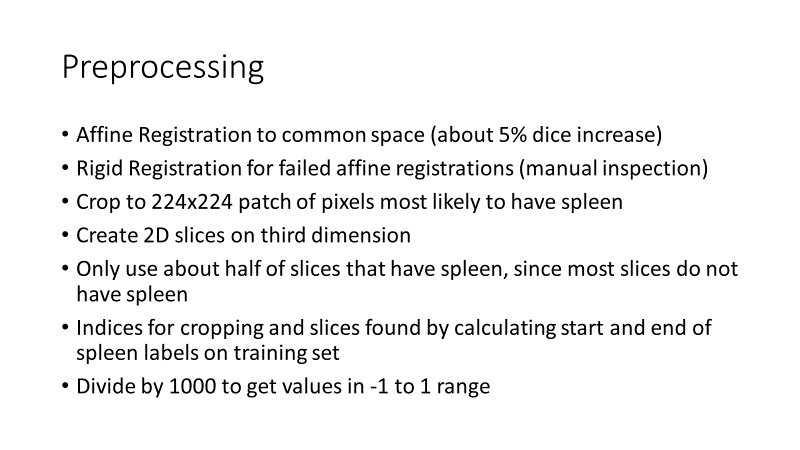
Slide 4



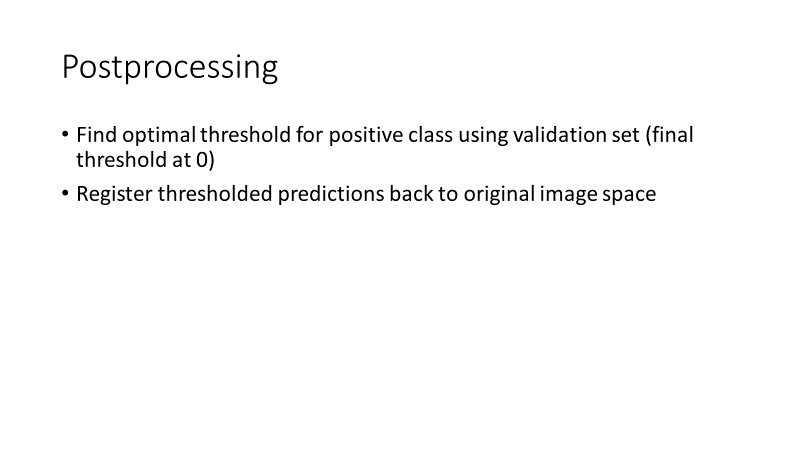
Slide 5



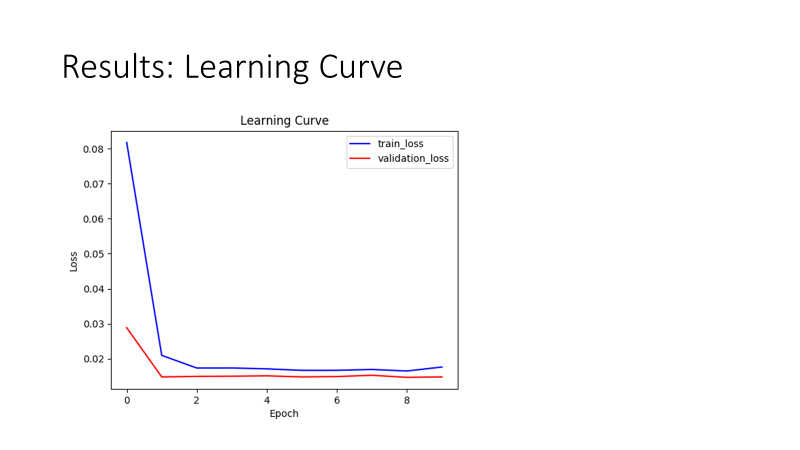
Slide 6



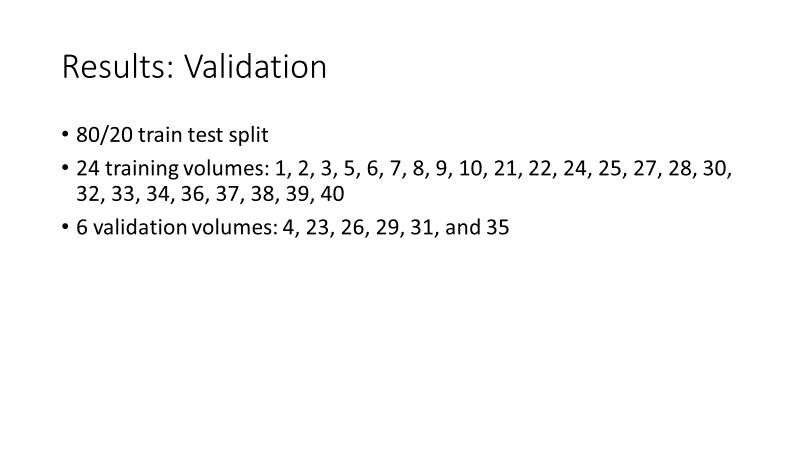
Slide 7



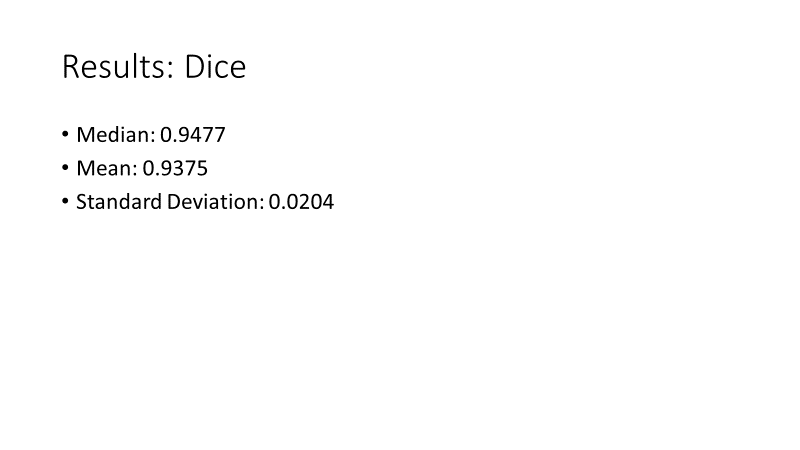
Slide 8



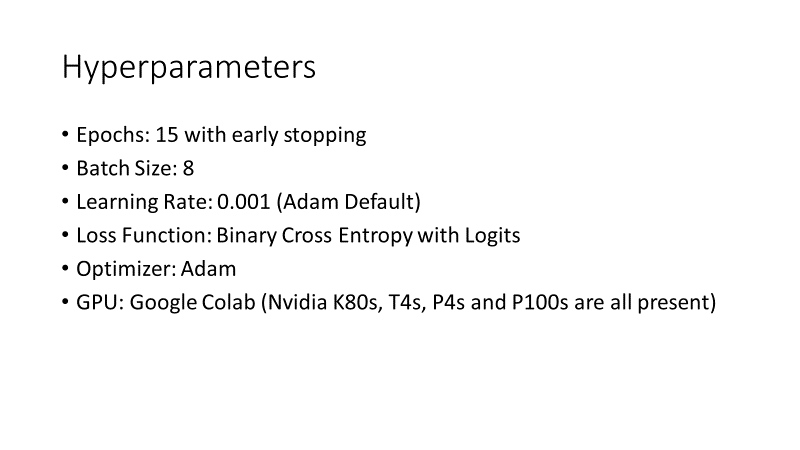
Slide 9



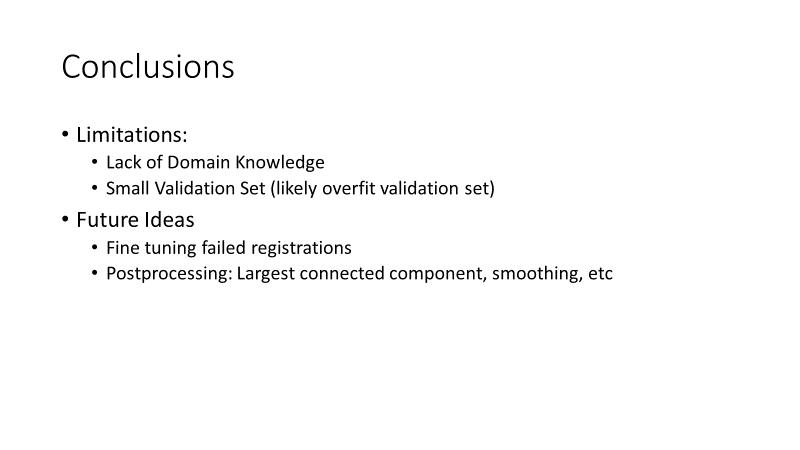
Slide 10



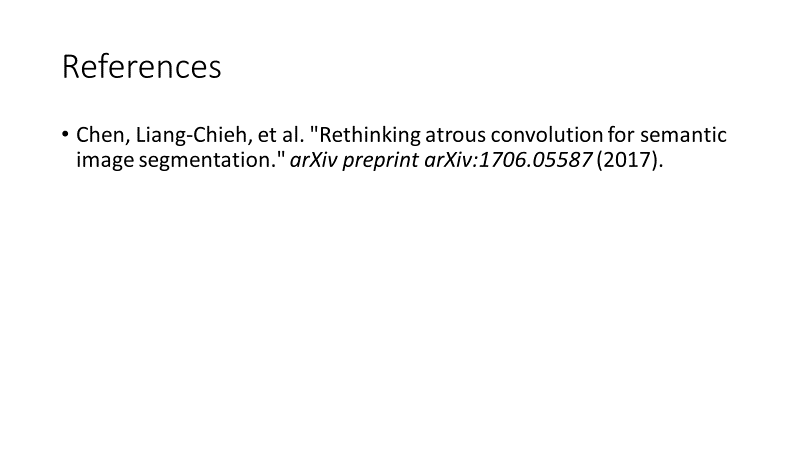
Slide 11



Slide 12



Slide 13



Code

preprocessing:

train\_val\_split.py

*# Perform 80/20 train-val split first***import** os  
**import** numpy **as** np  
**import** nibabel **as** nib  
**from** sklearn.model\_selection **import** train\_test\_split  
**import** shutil  
  
*# Constants*ORIGINAL\_LABELS = **"../../data/Original\_Training/label/"**TRAIN\_LABELS = **"../../data/Train/label/"**VAL\_LABELS = **"../../data/Val/label/"**ORIGINAL\_IMG = **"../../data/Original\_Training/img/"**TRAIN\_IMG = **"../../data/Train/img/"**VAL\_IMG = **"../../data/Val/img/"***# Get list of all the volume names*volumes = []  
**for** file\_name **in** os.listdir(ORIGINAL\_LABELS):  
 volumes.append(file\_name[5:])  
  
*# 80/20 train-val split with scikit learn*train, val = train\_test\_split(volumes, test\_size=0.2)  
  
*# Copy image and label to new directories for train and validation***for** volume **in** train:  
 *# Copy image* shutil.copy(ORIGINAL\_IMG + **"img"** + volume, TRAIN\_IMG + volume)  
 *# Copy label* shutil.copy(ORIGINAL\_LABELS + **"label"** + volume, TRAIN\_LABELS + volume)  
  
**for** volume **in** val:  
 *# Copy image* shutil.copy(ORIGINAL\_IMG + **"img"** + volume, VAL\_IMG + volume)  
 *# Copy label* shutil.copy(ORIGINAL\_LABELS + **"label"** + volume, VAL\_LABELS + volume)

register\_affine.py

*# Register all training volumes to 0007.nii.gz with affine registration.***import** ants  
**import** os  
  
*# Constants for path names*FIXED\_IMG = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Train/img/0007.nii.gz"**OLD\_TRAIN\_IMG = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Train/img/"**NEW\_TRAIN\_IMG = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Train/affine/img\_registered\_no\_resize/"**OLD\_TRAIN\_LABELS = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Train/label/"**NEW\_TRAIN\_LABELS = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Train/affine/label\_registered\_no\_resize/"**OLD\_VAL\_IMG = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Val/img/"**NEW\_VAL\_IMG = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Val/affine/img\_registered\_no\_resize/"**OLD\_VAL\_LABELS = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Val/label/"**NEW\_VAL\_LABELS = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Val/affine/label\_registered\_no\_resize/"***# Read in fixed image*fixed = ants.image\_read(FIXED\_IMG)  
  
*# Register all the training images***for** file\_name **in** os.listdir(OLD\_TRAIN\_IMG):  
 *# Read in moving image and corresponding label* moving\_image = ants.image\_read(OLD\_TRAIN\_IMG + file\_name)  
 label = ants.image\_read(OLD\_TRAIN\_LABELS + file\_name)  
 *# Calculate transform and apply to image and label. Save transformed image and label.* transform = ants.registration(fixed=fixed , moving=moving\_image,  
 type\_of\_transform=**'AffineFast'** )  
 transformed\_image = ants.apply\_transforms( fixed=fixed, moving=moving\_image,  
 transformlist=transform[**'fwdtransforms'**],  
 interpolator=**'nearestNeighbor'**)  
 transformed\_image.to\_file(NEW\_TRAIN\_IMG + file\_name)  
 transformed\_label = ants.apply\_transforms( fixed=fixed, moving=label,  
 transformlist=transform[**'fwdtransforms'**],  
 interpolator=**'nearestNeighbor'**)  
 transformed\_label.to\_file(NEW\_TRAIN\_LABELS + file\_name)  
  
*# Repeat for the validation images***for** file\_name **in** os.listdir(OLD\_VAL\_IMG):  
 *# Read in moving image and corresponding label* moving\_image = ants.image\_read(OLD\_VAL\_IMG + file\_name)  
 label = ants.image\_read(OLD\_VAL\_LABELS + file\_name)  
 *# Calculate transform and apply to image and label. Save transformed image and label.* transform = ants.registration(fixed=fixed , moving=moving\_image,  
 type\_of\_transform = **'AffineFast'** )  
 transformed\_image = ants.apply\_transforms( fixed=fixed, moving=moving\_image,  
 transformlist=transform[**'fwdtransforms'**],  
 interpolator = **'nearestNeighbor'**)  
 transformed\_image.to\_file(NEW\_VAL\_IMG + file\_name)  
 transformed\_label = ants.apply\_transforms( fixed=fixed, moving=label,  
 transformlist=transform[**'fwdtransforms'**],  
 interpolator = **'nearestNeighbor'**)  
 transformed\_label.to\_file(NEW\_VAL\_LABELS + file\_name)

register\_rigid.py

*# Register all training volumes to 0007.nii.gz. No resizing in this version***import** ants  
**import** os  
  
*# Constants for path names*FIXED\_IMG = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Train/img/0007.nii.gz"**OLD\_TRAIN\_IMG = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Train/img/"**NEW\_TRAIN\_IMG = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Train/rigid/img\_register\_rigid/"**OLD\_TRAIN\_LABELS = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Train/label/"**NEW\_TRAIN\_LABELS = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Train/rigid/label\_register\_rigid/"**OLD\_VAL\_IMG = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Val/img/"**NEW\_VAL\_IMG = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Val/rigid/img\_register\_rigid/"**OLD\_VAL\_LABELS = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Val/label/"**NEW\_VAL\_LABELS = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Val/rigid/label\_register\_rigid/"***# Read in fixed image.*fixed = ants.image\_read(FIXED\_IMG)  
  
*# Register all the training images***for** file\_name **in** os.listdir(OLD\_TRAIN\_IMG):  
 *# Read in moving image and corresponding label* moving\_image = ants.image\_read(OLD\_TRAIN\_IMG + file\_name)  
 label = ants.image\_read(OLD\_TRAIN\_LABELS + file\_name)  
 *# Calculate rigid transformation and apply to image and label* transform = ants.registration(fixed=fixed , moving=moving\_image,  
 type\_of\_transform=**'QuickRigid'** )  
 transformed\_image = ants.apply\_transforms( fixed=fixed, moving=moving\_image,  
 transformlist=transform[**'fwdtransforms'**],  
 interpolator=**'nearestNeighbor'**)  
 transformed\_image.to\_file(NEW\_TRAIN\_IMG + file\_name)  
 transformed\_label = ants.apply\_transforms( fixed=fixed, moving=label,  
 transformlist=transform[**'fwdtransforms'**],  
 interpolator=**'nearestNeighbor'**)  
 transformed\_label.to\_file(NEW\_TRAIN\_LABELS + file\_name)  
  
*# Repeat for the validation images***for** file\_name **in** os.listdir(OLD\_VAL\_IMG):  
 *# Read in moving image and corresponding label* moving\_image = ants.image\_read(OLD\_VAL\_IMG + file\_name)  
 label = ants.image\_read(OLD\_VAL\_LABELS + file\_name)  
 *# Calculate rigid transformation and apply to image and label* transform = ants.registration(fixed=fixed , moving=moving\_image,  
 type\_of\_transform = **'QuickRigid'** )  
 transformed\_image = ants.apply\_transforms( fixed=fixed, moving=moving\_image,  
 transformlist=transform[**'fwdtransforms'**],  
 interpolator = **'nearestNeighbor'**)  
 transformed\_image.to\_file(NEW\_VAL\_IMG + file\_name)  
 transformed\_label = ants.apply\_transforms( fixed=fixed, moving=label,  
 transformlist=transform[**'fwdtransforms'**],  
 interpolator = **'nearestNeighbor'**)  
 transformed\_label.to\_file(NEW\_VAL\_LABELS + file\_name)

calc\_stats.py

**import** nibabel **as** nib  
**import** numpy **as** np  
**import** os  
**import** matplotlib.pyplot **as** plt  
  
LABELS = **"../../data/Train/affine\_fixed/label\_registered/"***# Print out statistics for the registered images*print(**"######################################################"**)  
print(**"Statistics For Images at: "**, LABELS)  
print(**"######################################################"**)  
*# Lists for start and end indices of spleen for each image on each axis*min\_list\_z = []  
max\_list\_z = []  
min\_list\_x = []  
max\_list\_x = []  
min\_list\_y = []  
max\_list\_y = []  
**for** file\_name **in** os.listdir(LABELS):  
 *# Load the image* image = nib.load(LABELS + file\_name)  
 *# Get the array of values* image\_data = image.get\_fdata()  
  
 *# Filter by 1 values for spleen* spleen\_labels = np.where(image\_data == 1, 1, 0)  
  
 *# Print out the number of spleen labels* print(file\_name, **": "**, np.sum(spleen\_labels), **" total spleen labels"**)  
  
 *# Find where spleen starts and ends on z axis* spleen\_z = np.sum(spleen\_labels, axis=(0, 1))  
 spleen\_z = np.where(spleen\_z > 0, 1, 0)  
 spleen\_z\_sum = np.sum(spleen\_z)  
 *# Calculate the smallest and largest index with spleen label* spleen\_indices = np.nonzero(spleen\_z)  
 **if** spleen\_z\_sum > 0:  
 min\_spleen = np.min(spleen\_indices[0])  
 max\_spleen = np.max(spleen\_indices[0])  
 min\_list\_z.append(min\_spleen)  
 max\_list\_z.append(max\_spleen)  
  
 *# Find where spleen starts and ends on x axis* spleen\_x = np.sum(spleen\_labels, axis=(1, 2))  
 spleen\_x = np.where(spleen\_x > 0, 1, 0)  
 spleen\_x\_sum = np.sum(spleen\_x)  
 *# Calculate the smallest and largest index with spleen label* spleen\_indices = np.nonzero(spleen\_x)  
 **if** spleen\_x\_sum > 0:  
 min\_spleen = np.min(spleen\_indices[0])  
 max\_spleen = np.max(spleen\_indices[0])  
 min\_list\_x.append(min\_spleen)  
 max\_list\_x.append(max\_spleen)  
  
 *# Find where spleen starts and ends on y axis* spleen\_y = np.sum(spleen\_labels, axis=(0, 2))  
 spleen\_y = np.where(spleen\_y > 0, 1, 0)  
 spleen\_y\_sum = np.sum(spleen\_y)  
 *# Calculate the smallest and largest index with spleen label* spleen\_indices = np.nonzero(spleen\_y)  
 **if** spleen\_x\_sum > 0:  
 min\_spleen = np.min(spleen\_indices[0])  
 max\_spleen = np.max(spleen\_indices[0])  
 min\_list\_y.append(min\_spleen)  
 max\_list\_y.append(max\_spleen)  
  
*# Print out mean and standard deviation values for start and end of spleen in slices.*print(**"Mean z start slice for spleen: "**, np.mean(np.array(min\_list\_z)))  
print(**"Std z start slice for spleen: "**, np.std(np.array(min\_list\_z)))  
print(**"Smallest z start slice for spleen"**, np.min(np.array(min\_list\_z)))  
print(**"Largest z start slice for spleen"**, np.max(np.array(min\_list\_z)))  
print(**"Mean z end slice for spleen: "**, np.mean(np.array(max\_list\_z)))  
print(**"Std z end slice for spleen: "**, np.std(np.array(max\_list\_z)))  
print(**"Smallest z end slice for spleen"**, np.min(np.array(max\_list\_z)))  
print(**"Largest z end slice for spleen"**, np.max(np.array(max\_list\_z)))  
  
print(**"Mean x start slice for spleen: "**, np.mean(np.array(min\_list\_x)))  
print(**"Std x start slice for spleen: "**, np.std(np.array(min\_list\_x)))  
print(**"Smallest x start slice for spleen"**, np.min(np.array(min\_list\_x)))  
print(**"Largest x start slice for spleen"**, np.max(np.array(min\_list\_x)))  
print(**"Mean x end slice for spleen: "**, np.mean(np.array(max\_list\_x)))  
print(**"Std x end slice for spleen: "**, np.std(np.array(max\_list\_x)))  
print(**"Smallest x end slice for spleen"**, np.min(np.array(max\_list\_x)))  
print(**"Largest x end slice for spleen"**, np.max(np.array(max\_list\_x)))  
  
print(**"Mean y start slice for spleen: "**, np.mean(np.array(min\_list\_y)))  
print(**"Std y start slice for spleen: "**, np.std(np.array(min\_list\_y)))  
print(**"Smallest y start slice for spleen"**, np.min(np.array(min\_list\_y)))  
print(**"Largest y start slice for spleen"**, np.max(np.array(min\_list\_y)))  
print(**"Mean y end slice for spleen: "**, np.mean(np.array(max\_list\_y)))  
print(**"Std y end slice for spleen: "**, np.std(np.array(max\_list\_y)))  
print(**"Smallest y end slice for spleen"**, np.min(np.array(max\_list\_y)))  
print(**"Largest y end slice for spleen"**, np.max(np.array(max\_list\_y)))  
*# Plot the start and end slice distributions for z*plt.hist(np.array(min\_list\_z), bins=70)  
plt.show()  
plt.close()  
plt.hist(np.array(max\_list\_z), bins=70)  
plt.show()  
plt.close()  
*# Plot the start and end slice distributions for x*plt.hist(np.array(min\_list\_x), bins=70)  
plt.show()  
plt.close()  
plt.hist(np.array(max\_list\_x), bins=70)  
plt.show()  
plt.close()  
*# Plot the start and end slice distributions for y*plt.hist(np.array(min\_list\_y), bins=70)  
plt.show()  
plt.close()  
plt.hist(np.array(max\_list\_y), bins=70)  
plt.show()  
plt.close()

2d\_slice.py

*# Slice images into 2d slices for 2d networks.  
# Create filtered version of labels with only spleen labels.***import** nibabel **as** nib  
**import** numpy **as** np  
**import** os  
**from** skimage.transform **import** resize  
  
*# Constants for path names*NEW\_TRAIN\_LABELS\_FILTERED = **"../../../data/Train/affine\_fixed/label\_cropped\_filtered/"**OLD\_TRAIN\_LABELS = **"../../../data/Train/affine\_fixed/label\_registered/"**NEW\_TRAIN\_LABELS = **"../../../data/Train/affine\_fixed/label\_cropped/"**OLD\_TRAIN\_IMG = **"../../../data/Train/affine\_fixed/img\_registered/"**NEW\_TRAIN\_IMG = **"../../../data/Train/affine\_fixed/img\_cropped/"**NEW\_VAL\_LABELS\_FILTERED = **"../../../data/Val/affine\_fixed/label\_cropped\_filtered/"**OLD\_VAL\_LABELS = **"../../../data/Val/affine\_fixed/label\_registered/"**NEW\_VAL\_LABELS = **"../../../data/Val/affine\_fixed/label\_cropped/"**OLD\_VAL\_IMG = **"../../../data/Val/affine\_fixed/img\_registered/"**NEW\_VAL\_IMG = **"../../../data/Val/affine\_fixed/img\_cropped/"***# Start and end indices on z axis to reslice, since most slices do not have spleen*Z\_START = 75  
Z\_END = 145  
  
*# Start and end indices on x axis to reslice, since most slices do not have spleen*X\_START = 288  
X\_END = 512  
Y\_START = 110  
Y\_END = 334  
  
*# First for training set  
# Iterate through all the actual images***for** file\_name **in** os.listdir(OLD\_TRAIN\_IMG):  
 *# Load the image* image = nib.load(OLD\_TRAIN\_IMG + file\_name)  
 *# Get the array of values* image\_data = image.get\_fdata()  
 *# Fix zero values added by registration* image\_data = np.where(image\_data == 0.0, -1000, image\_data)  
 *# Divide by 1000 to normalize* image\_data = image\_data / 1000.0  
 *# Iterate through the third dimension to create 2d slices* **for** index **in** range(Z\_START, Z\_END + 1):  
 *# Check that we have not reached the end of the image* **if** index < image\_data.shape[2]:  
 *# Get the current slice* slice = image\_data[X\_START:X\_END, Y\_START:Y\_END, index]  
 *# Convert to float16* slice = slice.astype(np.float16)  
 *# Save as numpy array. Exclude extension prefix from file name.* np.save(NEW\_TRAIN\_IMG + file\_name[:-7] + **"\_"** + str(index), slice)  
  
*# Iterate through the labels  
# Store original and new sum of spleen labels for all training images*old\_train\_sum = 0  
new\_train\_sum = 0  
**for** file\_name **in** os.listdir(OLD\_TRAIN\_LABELS):  
 *# Load the image* image = nib.load(OLD\_TRAIN\_LABELS + file\_name)  
 *# Get the array of values* image\_data = image.get\_fdata()  
 *# Get version of labels with only spleen labels (label 1).* spleen = np.where(image\_data == 1, 1, 0)  
 *# Calculate original sum of spleen labels and increment running total* original\_spleen\_labels = np.sum(spleen)  
 old\_train\_sum += original\_spleen\_labels  
 *# Calculate new sum of spleen labels. Start at 0 and add at each slice* new\_spleen\_labels = 0  
 *# Iterate through the third dimension to create 2d slices* **for** index **in** range(Z\_START, Z\_END + 1):  
 *# Check that we have not reached the end of the image* **if** index < image\_data.shape[2]:  
 *# Get the current slice* slice = image\_data[X\_START:X\_END, Y\_START:Y\_END, index]  
 *# Convert to uint8* slice = slice.astype(np.uint8)  
 *# Save as numpy array. Exclude extension prefix from file name.* np.save(NEW\_TRAIN\_LABELS + file\_name[:-7] + **"\_"** + str(index), slice)  
 *# Save version of labels with only spleen labels (label 1).* spleen\_slice = np.where(slice==1, 1, 0)  
 spleen\_slice = spleen\_slice.astype(np.uint8)  
 np.save(NEW\_TRAIN\_LABELS\_FILTERED + file\_name[:-7] + **"\_"** + str(index), spleen\_slice)  
 *# Increment new sum of spleen labels* new\_spleen\_labels += np.sum(spleen\_slice)  
 *# Increment sum of all spleen labels* new\_train\_sum += new\_spleen\_labels  
print(**"Original Training Number of spleen labels: "**, old\_train\_sum)  
print(**"New Training Number of spleen labels: "**, new\_train\_sum)  
print(**"Percentage of Training Spleen Labels Retained: "**, new\_train\_sum / old\_train\_sum)  
  
print(**"Percentage of Training Labels that is spleen: "**, new\_train\_sum / (224\*224\*60\*24))  
print(**"Original percentage of Training Labels that is spleen: "**, new\_train\_sum / (512\*512\*163\*24))  
  
*# Repeat for Validation Set  
# Iterate through all the actual images***for** file\_name **in** os.listdir(OLD\_VAL\_IMG):  
 *# Load the image* image = nib.load(OLD\_VAL\_IMG + file\_name)  
 *# Get the array of values* image\_data = image.get\_fdata()  
 *# Fix zero values added by registration* image\_data = np.where(image\_data == 0.0, -1000, image\_data)  
 *# Divide by 1000 to normalize* image\_data = image\_data / 1000.0  
 *# Iterate through the third dimension to create 2d slices* **for** index **in** range(Z\_START, Z\_END + 1):  
 *# Check that we have not reached the end of the image* **if** index < image\_data.shape[2]:  
 *# Get the current slice* slice = image\_data[X\_START:X\_END, Y\_START:Y\_END, index]  
 *# Convert to float16* slice = slice.astype(np.float16)  
 *# Save as numpy array. Exclude extension prefix from file name.* np.save(NEW\_VAL\_IMG + file\_name[:-7] + **"\_"** + str(index), slice)  
  
*# Iterate through the labels  
# Store original and new sum of spleen labels for all validation images*old\_val\_sum = 0  
new\_val\_sum = 0  
**for** file\_name **in** os.listdir(OLD\_VAL\_LABELS):  
 *# Load the image* image = nib.load(OLD\_VAL\_LABELS + file\_name)  
 *# Get the array of values* image\_data = image.get\_fdata()  
 *# Get version of labels with only spleen labels (label 1).* spleen = np.where(image\_data == 1, 1, 0)  
 *# Calculate original sum of spleen labels* original\_spleen\_labels = np.sum(spleen)  
 old\_val\_sum += original\_spleen\_labels  
 *# Calculate new sum of spleen labels. Start at 0 and add at each slice* new\_spleen\_labels = 0  
 *# Iterate through the third dimension to create 2d slices* **for** index **in** range(Z\_START, Z\_END + 1):  
 *# Check that we have not reached the end of the image* **if** index < image\_data.shape[2]:  
 *# Get the current slice* slice = image\_data[X\_START:X\_END, Y\_START:Y\_END, index]  
 *# Convert to uint8* slice = slice.astype(np.uint8)  
 *# Save as numpy array. Exclude extension prefix from file name.* np.save(NEW\_VAL\_LABELS + file\_name[:-7] + **"\_"** + str(index), slice)  
 *# Save version of labels with only spleen labels (label 1).* spleen\_slice = np.where(slice==1, 1, 0)  
 spleen\_slice = spleen\_slice.astype(np.uint8)  
 np.save(NEW\_VAL\_LABELS\_FILTERED + file\_name[:-7] + **"\_"** + str(index), spleen\_slice)  
 *# Increment new sum of spleen labels* new\_spleen\_labels += np.sum(spleen\_slice)  
 *# Update sum of new spleen labels* new\_val\_sum += new\_spleen\_labels  
print(**"Original Validation Number of spleen labels: "**, old\_val\_sum)  
print(**"New Validation Number of spleen labels: "**, new\_val\_sum)  
print(**"Percentage of Spleen Labels Retained: "**, new\_val\_sum / old\_val\_sum)  
print(**"Percentage of Val Labels that is spleen: "**, new\_val\_sum / (224\*224\*60\*6))  
print(**"Original percentage of Val Labels that is spleen: "**, new\_val\_sum / (512\*512\*163\*6))

test\_preprocessing:

register\_affine.py

*# Register all testing volumes to 0007.nii.gz. No resizing in this version***import** ants  
**import** os  
  
*# Constants for path names*FIXED\_IMG = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Train/img/0007.nii.gz"**OLD\_TEST\_IMG = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Testing/img/"**NEW\_TEST\_IMG = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Testing/img\_registered\_affine/"***# Load in fixed image*fixed = ants.image\_read(FIXED\_IMG)  
  
*# Register all the testing images***for** file\_name **in** os.listdir(OLD\_TEST\_IMG):  
 *# Load in moving image* moving\_image = ants.image\_read(OLD\_TEST\_IMG + file\_name)  
 print(**"Registering "**, file\_name)  
 *# Perform registration* transform = ants.registration(fixed=fixed , moving=moving\_image,  
 type\_of\_transform=**'AffineFast'**, random\_seed=0)  
 transformed\_image = ants.apply\_transforms( fixed=fixed, moving=moving\_image,  
 transformlist=transform[**'fwdtransforms'**],  
 interpolator=**'nearestNeighbor'**)  
 *# Save transformed image* print(**"Saving "**, file\_name)  
 transformed\_image.to\_file(NEW\_TEST\_IMG + file\_name)

register\_rigid.py

*# Register all training volumes to 0007.nii.gz. No resizing in this version***import** ants  
**import** os  
  
*# Constants for path names*FIXED\_IMG = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Train/img/0007.nii.gz"**OLD\_TEST\_IMG = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Testing/img/"**NEW\_TEST\_IMG = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Testing/img\_registered\_rigid/"***# Load in fixed image*fixed = ants.image\_read(FIXED\_IMG)  
  
*# Register all the testing images***for** file\_name **in** os.listdir(OLD\_TEST\_IMG):  
 *# Load in moving image* moving\_image = ants.image\_read(OLD\_TEST\_IMG + file\_name)  
 print(**"Registering "**, file\_name)  
 *# Perform registration* transform = ants.registration(fixed=fixed , moving=moving\_image,  
 type\_of\_transform=**'QuickRigid'**, random\_seed=0)  
 transformed\_image = ants.apply\_transforms( fixed=fixed, moving=moving\_image,  
 transformlist=transform[**'fwdtransforms'**],  
 interpolator=**'nearestNeighbor'**)  
 *# Save transformed image* print(**"Saving "**, file\_name)  
 transformed\_image.to\_file(NEW\_TEST\_IMG + file\_name)

train.py

*# Author: Daniel Yan  
# Email: daniel.yan@vanderbilt.edu  
# Description: Train deeplabv3 for segmentation***import** argparse  
**from** matplotlib **import** pyplot **as** plt  
**import** numpy **as** np  
**import** pandas **as** pd  
**import** os  
**import** torch  
**from** torch.utils.data **import** Dataset, DataLoader  
**import** torch.nn **as** nn  
**import** torch.nn.functional **as** F  
**import** torch.optim **as** optim  
**from** torchvision **import** transforms, models  
**from** torch.optim.lr\_scheduler **import** StepLR  
**from** skimage **import** io  
  
*# Constants*MODEL\_NAME = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/bin/2d\_affine\_fixed/deeplabv3\_bce"**TRAIN\_IMG\_PATH = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Train/affine\_fixed/img\_cropped/"**TRAIN\_LABEL\_PATH = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Train/affine\_fixed/label\_cropped\_filtered/"**VAL\_IMG\_PATH = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Val/affine\_fixed/img\_cropped/"**VAL\_LABEL\_PATH = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Val/affine\_fixed/label\_cropped\_filtered/"***# Define dataset for image and segmentation mask***class** MyDataset(Dataset):  
 **def** \_\_init\_\_(self, image\_path, target\_path):  
 *# Create a list of all the names of the files to load* self.file\_names = list(os.listdir(image\_path))  
 *# Create list of images* self.images\_list = []  
 self.image\_names\_list = []  
 **for** file\_name **in** self.file\_names:  
 *# Load in image using numpy* image = np.load(image\_path + file\_name)  
 *# Convert to torch tensor* image\_tensor = torch.from\_numpy(image)  
 *# Insert first dimension for number of channels* image\_tensor = torch.unsqueeze(image\_tensor, 0)  
 image\_tensor\_expanded = image\_tensor.expand((3, 224, 224))  
 *# Add to list of images.* self.images\_list.append(image\_tensor\_expanded)  
 self.image\_names\_list.append(image\_path + file\_name)  
 *# Create list of target segmentations* self.targets\_list = []  
 self.target\_names\_list = []  
 **for** file\_name **in** list(os.listdir(image\_path)):  
 mask = np.load(target\_path + file\_name)  
 *# Convert to torch tensor* mask\_tensor = torch.from\_numpy(mask)  
 *# Add to list of masks.* self.targets\_list.append(mask\_tensor)  
 self.target\_names\_list.append(image\_path + file\_name)  
  
 **def** \_\_getitem\_\_(self, index):  
 **return** self.images\_list[index], self.targets\_list[index]  
  
 **def** \_\_len\_\_(self):  
 **return** len(self.images\_list)  
  
**def** train(model, device, train\_loader, optimizer, epoch, train\_losses):  
 *# Specify that we are in training phase* model.train()  
 *# Total Train Loss* total\_loss = 0  
 *# Iterate through all minibatches.* **for** index, (data, target) **in** enumerate(train\_loader):  
 *# Send training data and the training labels to GPU/CPU* data, target = data.to(device, dtype=torch.float32), target.to(device, dtype=torch.float32)  
 *# Zero the gradients carried over from previous step* optimizer.zero\_grad()  
 *# Obtain the predictions from forward propagation* output = model(data)[**"out"**]  
 output = torch.squeeze(output, 1)  
 *# Compute the cross entropy for the loss and update total loss.* loss = torch.nn.BCEWithLogitsLoss()(output, target)  
 total\_loss += loss.item()  
 *# Perform backward propagation to compute the negative gradient, and  
 # update the gradients with optimizer.step()* loss.backward()  
 optimizer.step()  
  
 *# Update training error and add to accumulation of training loss over time.* train\_error = total\_loss / len(train\_loader)  
 train\_losses.append(train\_error)  
 *# Print out the epoch and train loss* print(**"############################################################################"**)  
 print(**"Train Epoch: "**, epoch)  
 print(**"############################################################################"**)  
 print(**"Average Training Loss: "**, train\_error)  
 **return** train\_losses  
  
  
**def** test(model, device, test\_loader, test\_losses):  
 *# Create dictionary of predictions for different thresholds.  
 # Initialize sums of true positives, true negatives, false positives, and false negatives to 0* threshold\_dict = {}  
 **for** threshold **in** [0.001, 0.01, 0.1, 0.25, 0.5]:  
 threshold\_dict[str(threshold)] = {}  
 threshold\_dict[str(threshold)][**"total\_tp"**] = 0  
 threshold\_dict[str(threshold)][**"total\_tn"**] = 0  
 threshold\_dict[str(threshold)][**"total\_fp"**] = 0  
 threshold\_dict[str(threshold)][**"total\_fn"**] = 0  
 *# Specify that we are in evaluation phase* model.eval()  
 *# Set the loss and number of correct instances initially to 0.* test\_loss = 0  
 *# No gradient calculation because we are in testing phase.* **with** torch.no\_grad():  
 *# For each testing example, we run forward  
 # propagation to calculate the  
 # testing prediction. Update the total loss  
 # and f1 score with counters from above* **for** index, (data, target) **in** enumerate(test\_loader):  
 *# Send training data and the training labels to GPU/CPU* data, target = data.to(device, dtype=torch.float32), target.to(device, dtype=torch.float32)  
 *# Obtain the output from the model* output = model(data)[**"out"**]  
 output = torch.squeeze(output, 1)  
 *# Calculate the loss using cross entropy.* loss = torch.nn.BCEWithLogitsLoss()(output, target)  
 *# Increment the total test loss* test\_loss += loss.item()  
  
 *# Convert output to numpy array* output = output.cpu().numpy()  
 *# Calculate stats for each threshold* **for** threshold **in** [0.001, 0.01, 0.1, 0.25, 0.5]:  
 *# Filter both the prediction and the target by only class 1 for spleen* pred\_filtered = np.where(output > threshold, 1, 0)  
 target\_filtered = np.where(target.cpu().numpy() == 1, 1, 0)  
  
 *# Calculate the true positives, false positives, true negatives, and false negatives  
 # and increment total sums* true\_positives = float(np.sum(np.where(np.logical\_and(pred\_filtered == 1, target\_filtered == 1), 1, 0)))  
 false\_positives = float(np.sum(np.where(np.logical\_and(pred\_filtered == 1, target\_filtered == 0), 1, 0)))  
 true\_negatives = float(np.sum(np.where(np.logical\_and(pred\_filtered == 0, target\_filtered == 0), 1, 0)))  
 false\_negatives = float(np.sum(np.where(np.logical\_and(pred\_filtered == 0, target\_filtered == 1), 1, 0)))  
 threshold\_dict[str(threshold)][**"total\_tp"**] += true\_positives  
 threshold\_dict[str(threshold)][**"total\_tn"**] += true\_negatives  
 threshold\_dict[str(threshold)][**"total\_fp"**] += false\_positives  
 threshold\_dict[str(threshold)][**"total\_fn"**] += false\_negatives  
  
 *# Calculate precision, recall, and f1 and print out statistics for validation set* print(**"Average Validation Loss: "**, test\_loss / len(test\_loader))  
 *# Find results for each threshold.* **for** threshold **in** [0.001, 0.01, 0.1, 0.25, 0.5]:  
 print(**"At threshold "**, str(threshold))  
 print(**"Total Validation True Positives: "**, threshold\_dict[str(threshold)][**"total\_tp"**])  
 print(**"Total Validation True Negatives: "**, threshold\_dict[str(threshold)][**"total\_tn"**])  
 print(**"Total Validation False Positives: "**, threshold\_dict[str(threshold)][**"total\_fp"**])  
 print(**"Total Validation False Negatives: "**, threshold\_dict[str(threshold)][**"total\_fn"**])  
  
 *# Calculate precision and recall and F1* **if** (threshold\_dict[str(threshold)][**"total\_tp"**] > 0 **and** threshold\_dict[str(threshold)][**"total\_fp"**]> 0  
 **and** threshold\_dict[str(threshold)][**"total\_fn"**] > 0):  
 precision = threshold\_dict[str(threshold)][**"total\_tp"**] / (threshold\_dict[str(threshold)][**"total\_tp"**]  
 + threshold\_dict[str(threshold)][**"total\_fp"**])  
 recall = threshold\_dict[str(threshold)][**"total\_tp"**] / (threshold\_dict[str(threshold)][**"total\_tp"**] +  
 threshold\_dict[str(threshold)][**"total\_fn"**])  
 f1 = 2 \* precision \* recall / (precision + recall)  
 print(**"Precision: "**, precision)  
 print(**"Recall: "**, recall)  
 print(**"F1: "**, f1)  
  
 *# Append test loss to total losses* test\_losses.append(test\_loss / len(test\_loader))  
 **return** test\_losses  
  
*# Main structure***def** main():  
 print(**"Entering Main"**)  
 *# Command line arguments for hyperparameters of model/training.* parser = argparse.ArgumentParser(description=**'PyTorch Object Detection'**)  
 parser.add\_argument(**'--batch-size'**, type=int, default=8, metavar=**'N'**,  
 help=**'input batch size for training (default: 8)'**)  
 parser.add\_argument(**'--test-batch-size'**, type=int, default=8, metavar=**'N'**,  
 help=**'input batch size for testing (default: 8)'**)  
 parser.add\_argument(**'--epochs'**, type=int, default=50, metavar=**'N'**,  
 help=**'number of epochs to train (default: 50)'**)  
 parser.add\_argument(**'--lr'**, type=float, default=0.001, metavar=**'LR'**,  
 help=**'learning rate (default: 0.001)'**)  
 parser.add\_argument(**'--no-cuda'**, action=**'store\_true'**, default=**False**,  
 help=**'disables CUDA training'**)  
 parser.add\_argument(**'--seed'**, type=int, default=1, metavar=**'S'**,  
 help=**'random seed (default: 1)'**)  
 args = parser.parse\_args()  
 *# Command to use gpu depending on command line arguments and if there is a cuda device* use\_cuda = **not** args.no\_cuda **and** torch.cuda.is\_available()  
  
 *# Random seed to use* torch.manual\_seed(args.seed)  
  
 *# Set to either use gpu or cpu* device = torch.device(**"cuda" if** use\_cuda **else "cpu"**)  
  
 *# GPU keywords.* kwargs = {**'num\_workers'**: 1, **'pin\_memory'**: **True**} **if** use\_cuda **else** {}  
  
 *# Load in the dataset* train\_data = MyDataset(image\_path=TRAIN\_IMG\_PATH, target\_path=TRAIN\_LABEL\_PATH)  
 val\_data = MyDataset(image\_path=VAL\_IMG\_PATH, target\_path=VAL\_LABEL\_PATH)  
 *# Create data loader for training and validation* train\_loader = DataLoader(train\_data, batch\_size=args.batch\_size, shuffle=**True**, num\_workers=0, drop\_last=**True**)  
 val\_loader = DataLoader(val\_data, batch\_size=args.test\_batch\_size, shuffle=**False**, num\_workers=0)  
 print(**"Finished Loading Data"**)  
  
 *# Send model to gpu* model = models.segmentation.deeplabv3\_resnet101(num\_classes=1).to(device)  
 *# Specify Adam optimizer* optimizer = optim.Adam(model.parameters(), lr=args.lr)  
  
 *# Store training and validation losses over time* train\_losses = []  
 val\_losses = []  
  
 *# Create scheduler.* scheduler = StepLR(optimizer, step\_size=1)  
  
 *# Store the lowest loss found so far for early stopping* lowest\_loss = 1000  
  
 *# Train the model for the set number of epochs* **for** epoch **in** range(1, args.epochs + 1):  
 *# Train and validate for this epoch* train\_losses = train(model, device, train\_loader, optimizer, epoch, train\_losses)  
 val\_losses = test(model, device, val\_loader, val\_losses)  
 scheduler.step()  
 *# Create learning curve* figure, axes = plt.subplots()  
 *# Set axes labels and title* axes.set(xlabel=**"Epoch"**, ylabel=**"Loss"**, title=**"Learning Curve"**)  
 *# Plot the learning curves for training and validation loss* axes.plot(np.array(train\_losses), label=**"train\_loss"**, c=**"b"**)  
 axes.plot(np.array(val\_losses), label=**"validation\_loss"**, c=**"r"**)  
 plt.legend()  
 *# Save the figure* plt.savefig(MODEL\_NAME + **".png"**)  
 plt.close()  
  
 *# If we find the lowest loss so far, store the model and learning curve* **if** lowest\_loss > val\_losses[epoch - 1]:  
 *# Update the lowest loss* lowest\_loss = val\_losses[epoch - 1]  
 print(**"New lowest validation loss: "**, lowest\_loss)  
  
 *# Save the model* torch.save(model.state\_dict(), MODEL\_NAME + **".pt"**)  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()

calc\_val\_volumes.py

*# Generate predictions for the volumes using the model.***import** nibabel **as** nib  
**import** numpy **as** np  
**import** os  
**import** torch  
**from** torchvision **import** transforms, models  
  
  
VAL\_IMG\_PATH = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Val/affine\_fixed/img\_registered/"**SAVE\_VOL\_PATH = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/results/Val/affine\_fixed/prediction\_float/"**MODEL\_NAME = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/bin/2d\_affine\_fixed/deeplabv3\_bce\_resnet50.pt"***# Start and end indices for where to slice on each axis*X\_START = 288  
X\_END = 512  
Y\_START = 110  
Y\_END = 334  
Z\_START = 75  
Z\_END = 145  
  
**def** main():  
 *# Set to either use gpu or cpu* device = torch.device(**"cuda"**)  
  
 *# GPU keywords.* kwargs = {**'num\_workers'**: 1, **'pin\_memory'**: **True**}  
  
 *# Load in saved model* model = models.segmentation.deeplabv3\_resnet50(num\_classes=1).to(device)  
 model.load\_state\_dict(torch.load(MODEL\_NAME))  
  
 *# Specify that we are in evaluation phase* model.eval()  
  
 *# No gradient calculation because we are in testing phase.* **with** torch.no\_grad():  
 *# Iterate through all validation volumes and calculate predicted segmentations for each one.* **for** file\_name **in** os.listdir(VAL\_IMG\_PATH):  
 print(**"Calculating Predictions for"**, file\_name)  
 *# Load the image* image = nib.load(VAL\_IMG\_PATH + file\_name)  
 *# Get the array of values* image\_data = image.get\_fdata()  
 *# Fix zero values added by registration* image\_data = np.where(image\_data == 0.0, -1000, image\_data)  
 *# Divide by 1000 to normalize* image\_data = image\_data / 1000.0  
 *# Slice for where the spleen is present* spleen = image\_data[X\_START:X\_END, Y\_START:Y\_END, Z\_START:Z\_END]  
 *# Put the z axis on the first dimension since each z slice  
 # represents a separate image in our 2D model* spleen = np.transpose(spleen, (2, 0, 1))  
 *# Convert to torch tensor* spleen\_tensor = torch.from\_numpy(spleen)  
 *# Insert dimension for number of channels* spleen\_tensor = torch.unsqueeze(spleen\_tensor, 1)  
 *# Expand to 3 channels for deeplabv3 architecture* spleen\_tensor = spleen\_tensor.expand((Z\_END-Z\_START, 3, X\_END-X\_START, Y\_END-Y\_START))  
 spleen\_tensor = spleen\_tensor.to(device, dtype=torch.float32)  
  
 *# Calculate the segmentation output from the model* segmentation = model(spleen\_tensor)[**"out"**]  
 *# Take out the dimension for number of channels* segmentation = torch.squeeze(segmentation, 1)  
 *# Convert to numpy array and transpose again* segmentation\_np = segmentation.cpu().numpy()  
 segmentation\_np = np.transpose(segmentation\_np, (1, 2, 0))  
  
 *# Create new numpy array of -10 values and insert in area of predictions* prediction = np.ones(image\_data.shape) \* -10  
 prediction[X\_START:X\_END, Y\_START:Y\_END, Z\_START:Z\_END] = segmentation\_np  
  
 *# Save the prediction to file.* output = nib.Nifti1Image(prediction, image.affine)  
 nib.save(output, SAVE\_VOL\_PATH + file\_name)  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()

calc\_test\_volumes.py

*# Generate predictions for the volumes using the model.***import** nibabel **as** nib  
**import** numpy **as** np  
**import** os  
**import** torch  
**from** torchvision **import** models  
  
  
VAL\_IMG\_PATH = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Testing/img\_registered\_all/"**SAVE\_VOL\_PATH = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/results/Testing/prediction\_float/"**MODEL\_NAME = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/bin/2d\_affine\_fixed/deeplabv3\_bce\_resnet50.pt"***# Start and end indices for where to slice on each axis*X\_START = 288  
X\_END = 512  
Y\_START = 110  
Y\_END = 334  
Z\_START = 75  
Z\_END = 145  
  
**def** main():  
 *# Set to either use gpu or cpu* device = torch.device(**"cuda"**)  
  
 *# GPU keywords.* kwargs = {**'num\_workers'**: 1, **'pin\_memory'**: **True**}  
  
 *# Load in saved model* model = models.segmentation.deeplabv3\_resnet50(num\_classes=1).to(device)  
 model.load\_state\_dict(torch.load(MODEL\_NAME))  
  
 *# Specify that we are in evaluation phase* model.eval()  
  
 *# No gradient calculation because we are in testing phase.* **with** torch.no\_grad():  
 *# Iterate through all validation volumes and calculate predicted segmentations for each one.* **for** file\_name **in** os.listdir(VAL\_IMG\_PATH):  
 print(**"Calculating Predictions for"**, file\_name)  
 *# Load the image* image = nib.load(VAL\_IMG\_PATH + file\_name)  
 *# Get the array of values* image\_data = image.get\_fdata()  
 *# Fix zero values added by registration* image\_data = np.where(image\_data == 0.0, -1000, image\_data)  
 *# Divide by 1000 to normalize* image\_data = image\_data / 1000.0  
 *# Slice for where the spleen is present* spleen = image\_data[X\_START:X\_END, Y\_START:Y\_END, Z\_START:Z\_END]  
 *# Put the z axis on the first dimension since each z slice  
 # represents a separate image in our 2D model* spleen = np.transpose(spleen, (2, 0, 1))  
 *# Convert to torch tensor* spleen\_tensor = torch.from\_numpy(spleen)  
 *# Insert dimension for number of channels* spleen\_tensor = torch.unsqueeze(spleen\_tensor, 1)  
 *# Expand to 3 channels for deeplabv3 architecture* spleen\_tensor = spleen\_tensor.expand((Z\_END-Z\_START, 3, X\_END-X\_START, Y\_END-Y\_START))  
 spleen\_tensor = spleen\_tensor.to(device, dtype=torch.float32)  
  
 *# Calculate the segmentation output from the model* segmentation = model(spleen\_tensor)[**"out"**]  
 *# Take out the dimension for number of channels* segmentation = torch.squeeze(segmentation, 1)  
 *# Convert to numpy array and transpose again* segmentation\_np = segmentation.cpu().numpy()  
 segmentation\_np = np.transpose(segmentation\_np, (1, 2, 0))  
  
 *# Create new numpy array of -10 values for the numpy output,  
 # and insert the predicted area.* prediction = np.ones(image\_data.shape) \* -10  
 prediction[X\_START:X\_END, Y\_START:Y\_END, Z\_START:Z\_END] = segmentation\_np  
  
 *# Save the prediction to file.* output = nib.Nifti1Image(prediction, image.affine)  
 nib.save(output, SAVE\_VOL\_PATH + file\_name)  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()

postprocessing:

calc\_threshold.py

*# Calculate F1 score at different thresholds.***import** nibabel **as** nib  
**import** numpy **as** np  
**import** os  
  
*# Path for the predicted volumes*PREDICTION\_PATH = **"../../results/Val/affine\_fixed/prediction\_float/"***# Path for the actual labels*LABELS\_PATH = **"../../data/Val/affine\_fixed/label\_registered/"***# Path to save the thresholded volumes*THRESHOLD\_PATH = **"../../results/Val/affine\_fixed/prediction\_thresholded/"  
  
def** main():  
 *# Step 1: Find the best threshold for counting a prediction as class 1  
  
 # Best threshold and f1 so far* best\_threshold = 0  
 best\_f1 = 0  
 *# Iterate through different thresholds to calculate f1 at each threshold* **for** threshold **in** [-0.5, -0.25, -0.1, 0, 0.1, 0.25, 0.5]:  
 *# List of precision, recall, and f1 scores* precision\_list = []  
 recall\_list = []  
 f1\_list = []  
 *# Iterate through all validation volumes and calculate results at different thresholds for each one* **for** file\_name **in** os.listdir(PREDICTION\_PATH):  
 *# Load in the actual labels* label = nib.load(LABELS\_PATH + file\_name)  
 label = label.get\_fdata()  
 *# Filter for only spleen labels* label = np.where(label == 1, 1, 0)  
  
 *# Load the prediction* image = nib.load(PREDICTION\_PATH + file\_name)  
 *# Get the array of values* image\_data = image.get\_fdata()  
 *# Threshold for predictions* prediction = np.where(image\_data >= threshold, 1, 0)  
  
 *# Calculate true positives, false positives, and false negatives* tp = np.where(np.logical\_and(label == 1, prediction == 1))  
 tp = np.sum(tp)  
 fp = np.where(np.logical\_and(label == 0, prediction == 1))  
 fp = np.sum(fp)  
 fn = np.where(np.logical\_and(label == 1, prediction == 0))  
 fn = np.sum(fn)  
  
 *# Calculate precision, recall, and f1* precision = tp / (tp + fp)  
 recall = tp / (tp + fn)  
 f1 = 2 \* precision \* recall / (precision + recall)  
  
 *# Add to list of precision, recall, f1* precision\_list.append(precision)  
 recall\_list.append(recall)  
 f1\_list.append(f1)  
  
 *# Print precision, recall, f1 at the threshold* f1 = np.mean(np.array(f1\_list))  
 precision = np.mean(np.array(precision\_list))  
 recall = np.mean(np.array(recall\_list))  
 print(**"##########################################################################"**)  
 print(**"Threshold: "**, threshold)  
 print(**"##########################################################################"**)  
 print(**"Precision: "**, precision\_list)  
 print(**"Precision Mean: "**, precision)  
 print(**"Recall: "**, recall\_list)  
 print(**"Recall Mean: "**, recall)  
 print(**"F1: "**, f1\_list)  
 print(**"F1 Mean: "**, f1)  
  
 *# Check if this threshold is the best f1 score so far* **if** f1 > best\_f1:  
 best\_f1 = f1  
 best\_threshold = threshold  
  
  
 *# Step 2: Save the predictions at the best threshold* print(**"##########################################################################"**)  
 print(**"Saving the Predictions at the Best Threshold of "**, best\_threshold)  
 print(**"##########################################################################"**)  
 *# Iterate through all validation volumes and calculate results at different thresholds for each one* **for** file\_name **in** os.listdir(PREDICTION\_PATH):  
 *# Load the prediction* image = nib.load(PREDICTION\_PATH + file\_name)  
 *# Get the array of values* image\_data = image.get\_fdata()  
 *# Threshold for predictions* prediction = np.where(image\_data >= best\_threshold, 1, 0)  
 prediction = nib.Nifti1Image(prediction, image.affine)  
 *# Save the prediction* nib.save(prediction, THRESHOLD\_PATH + file\_name)  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()

calc\_stats.py

*# Calculate statistics back in original space***import** nibabel **as** nib  
**import** numpy **as** np  
**import** os  
  
*# Constants for path names*ACTUAL\_LABEL = **"../../data/Val/label/"**PREDICTED\_LABEL = **"../../results/Val/affine\_fixed/deregistered/"***# List of precision, recall, and f1 scores*precision\_list = []  
recall\_list = []  
f1\_list = []  
  
*# Transform all validation predictions back to original space***for** file\_name **in** os.listdir(PREDICTED\_LABEL):  
 *# Load in the actual labels* actual = nib.load(ACTUAL\_LABEL + file\_name)  
 actual = actual.get\_fdata()  
 *# Filter for only spleen labels* actual = np.where(actual == 1, 1, 0)  
  
 *# Load in predicted labels* predicted = nib.load(PREDICTED\_LABEL + file\_name)  
 predicted = predicted.get\_fdata()  
  
 *# Calculate true positives, false positives, and false negatives* tp = np.where(np.logical\_and(actual == 1, predicted == 1))  
 tp = np.sum(tp)  
 fp = np.where(np.logical\_and(actual == 0, predicted == 1))  
 fp = np.sum(fp)  
 fn = np.where(np.logical\_and(actual == 1, predicted == 0))  
 fn = np.sum(fn)  
  
 *# Calculate precision, recall, and f1* precision = tp / (tp + fp)  
 recall = tp / (tp + fn)  
 f1 = 2 \* precision \* recall / (precision + recall)  
  
 *# Add to list of precision, recall, f1* precision\_list.append(precision)  
 recall\_list.append(recall)  
 f1\_list.append(f1)  
  
print(**"Precision: "**, precision\_list)  
print(**"Recall: "**, recall\_list)  
print(**"F1: "**, f1\_list)  
print(**"Mean Precision: "**, np.mean(np.array(precision\_list)))  
print(**"Mean Recall: "**, np.mean(np.array(recall\_list)))  
print(**"Mean F1: "**, np.mean(np.array(f1\_list)))  
print(**"Std F1: "**, np.std(np.array(f1\_list)))  
print(**"Median F1: "**, np.median(np.array(f1\_list)))

deregister.py

*# Register the labels back to original space***import** ants  
**import** os  
*# Constants for path names*VAL\_IMG = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Val/img/"**VAL\_IMG\_REGISTER = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Val/affine\_fixed/img\_registered/"**VAL\_PREDICTIONS = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/results/Val/affine\_fixed/prediction\_thresholded/"**NEW\_VAL\_PREDICTIONS = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/results/Val/affine\_fixed/deregistered/"***# Transform all validation predictions back to original space***for** file\_name **in** os.listdir(VAL\_IMG):  
 print(**"Deregistering: "**, file\_name)  
 *# Read in fixed image as the original image and the moving image as the  
 # registered image, as well as the label to transform* fixed = ants.image\_read(VAL\_IMG + file\_name)  
 moving\_image = ants.image\_read(VAL\_IMG\_REGISTER + file\_name)  
 label = ants.image\_read(VAL\_PREDICTIONS + file\_name)  
 *# Affine register, except for 0004.nii.gz, which was registered with rigid* **if** file\_name != **"0004.nii.gz"**:  
 transform = ants.registration(fixed=fixed, moving=moving\_image,  
 type\_of\_transform = **'Affine'**, random\_seed=0)  
 **else**:  
 transform = ants.registration(fixed=fixed, moving=moving\_image,  
 type\_of\_transform = **'Rigid'**, random\_seed=0)  
 *# Apply transform for label back to original space* deregistered\_label = ants.apply\_transforms(fixed=fixed, moving=label,  
 transformlist=transform[**'fwdtransforms'**],  
 interpolator=**'nearestNeighbor'**)  
 deregistered\_label.to\_file(NEW\_VAL\_PREDICTIONS + file\_name)

test\_postprocessing

threshold.py

*# Threshold at 0.01, which was the threshold used for validation set.***import** nibabel **as** nib  
**import** numpy **as** np  
**import** os  
  
PREDICTION\_PATH = **"../../results/Testing/prediction\_float/"***# Path to save the thresholded volumes*THRESHOLD\_PATH = **"../../results/Testing/prediction\_thresholded/"***# Threshold for prediction of 1 value found with validation set*THRESHOLD = 0  
  
**def** main():  
 *# Iterate through all validation volumes and calculate results at different thresholds for each one* **for** file\_name **in** os.listdir(PREDICTION\_PATH):  
 *# Load the prediction* image = nib.load(PREDICTION\_PATH + file\_name)  
 *# Get the array of values* image\_data = image.get\_fdata()  
 *# Threshold for predictions* prediction = np.where(image\_data >= THRESHOLD, 1, 0)  
 prediction = nib.Nifti1Image(prediction, image.affine)  
 *# Save the prediction* nib.save(prediction, THRESHOLD\_PATH + file\_name)  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()

deregister.py

*# Register the labels back to original space***import** ants  
**import** os  
*# Constants for path names*VAL\_IMG = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Testing/img/"**VAL\_IMG\_REGISTER = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/data/Testing/img\_registered\_all/"**VAL\_PREDICTIONS = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/results/Testing/prediction\_thresholded/"**NEW\_VAL\_PREDICTIONS = **"/content/drive/My Drive/cs8395\_deep\_learning/assignment3/results/Testing/prediction\_final/"***# Transform all validation predictions back to original space***for** file\_name **in** os.listdir(VAL\_IMG):  
 *# Read in fixed image as the original image and the moving image as the  
 # registered image, as well as the label to transform* print(**"Deregistering: "**, file\_name)  
 fixed = ants.image\_read(VAL\_IMG + file\_name)  
 moving\_image = ants.image\_read(VAL\_IMG\_REGISTER + file\_name)  
 label = ants.image\_read(VAL\_PREDICTIONS + file\_name)  
 *# Affine register, except for 0066.nii.gz, which was registered with rigid* **if** file\_name != **"0066.nii.gz"**:  
 transform = ants.registration(fixed=fixed, moving=moving\_image,  
 type\_of\_transform = **'Affine'**, random\_seed=0)  
 **else**:  
 transform = ants.registration(fixed=fixed, moving=moving\_image,  
 type\_of\_transform = **'Rigid'**, random\_seed=0)  
 *# Apply transform for label back to original space* deregistered\_label = ants.apply\_transforms(fixed=fixed, moving=label,  
 transformlist=transform[**'fwdtransforms'**],  
 interpolator=**'nearestNeighbor'**)  
 deregistered\_label.to\_file(NEW\_VAL\_PREDICTIONS + file\_name)