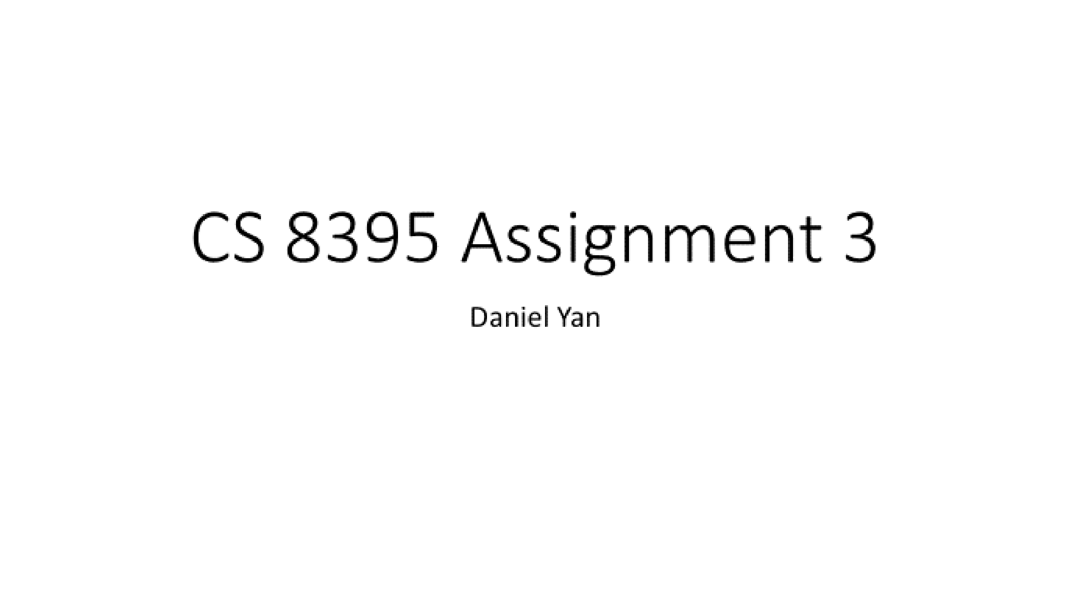
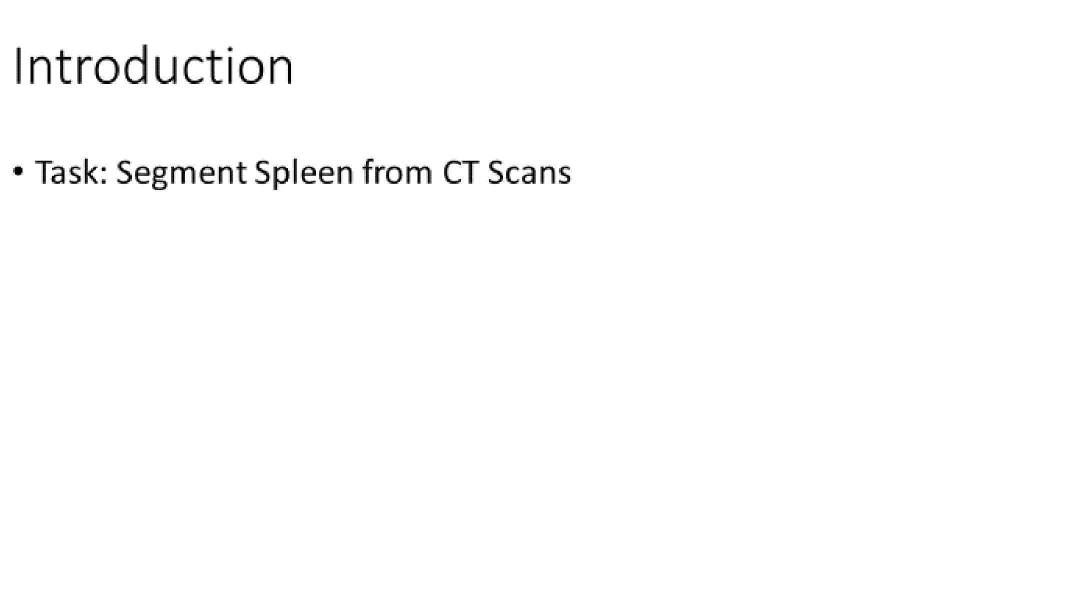
Slide 1



CS 8395 Assignment 3

Daniel Yan

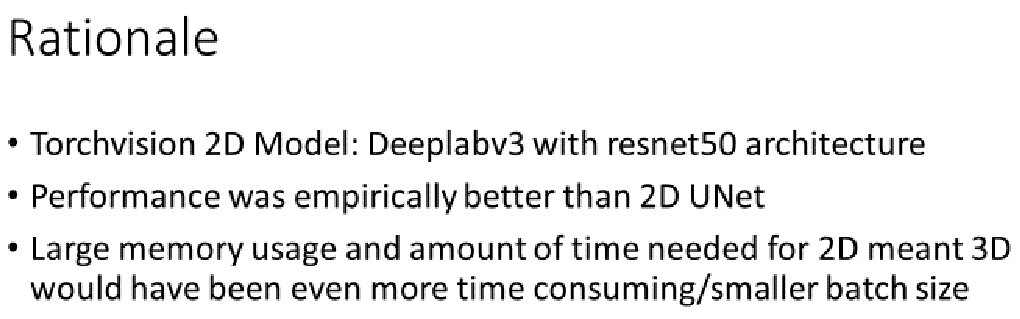
Slide 2



Introduction

- Task: Segment Spleen from CT Scans

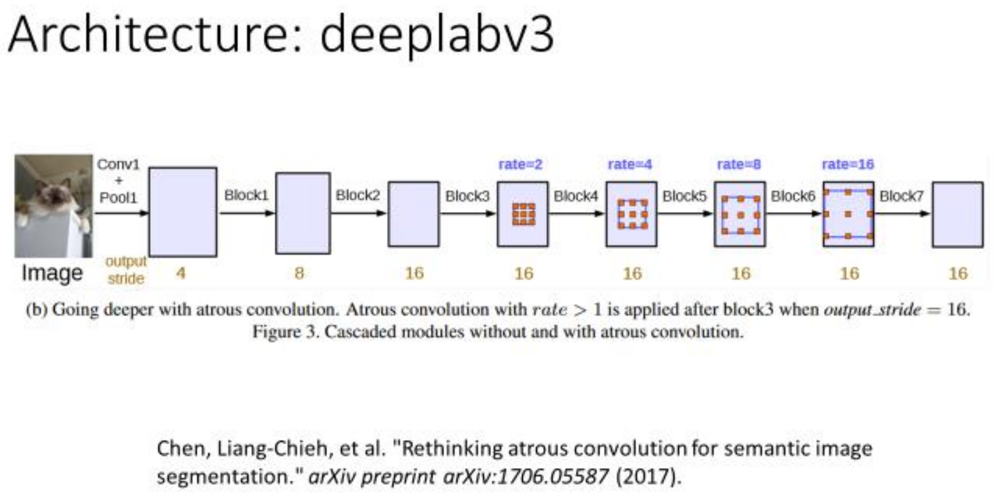
Slide 3



Rationale

- Torchvision 2D Model: Deeplabv3 with resnet50 architecture
- Performance was empirically better than 2D UNet
- Large memory usage and amount of time needed for 2D meant 3D would have been even more time consuming/smaller batch size

Slide 4

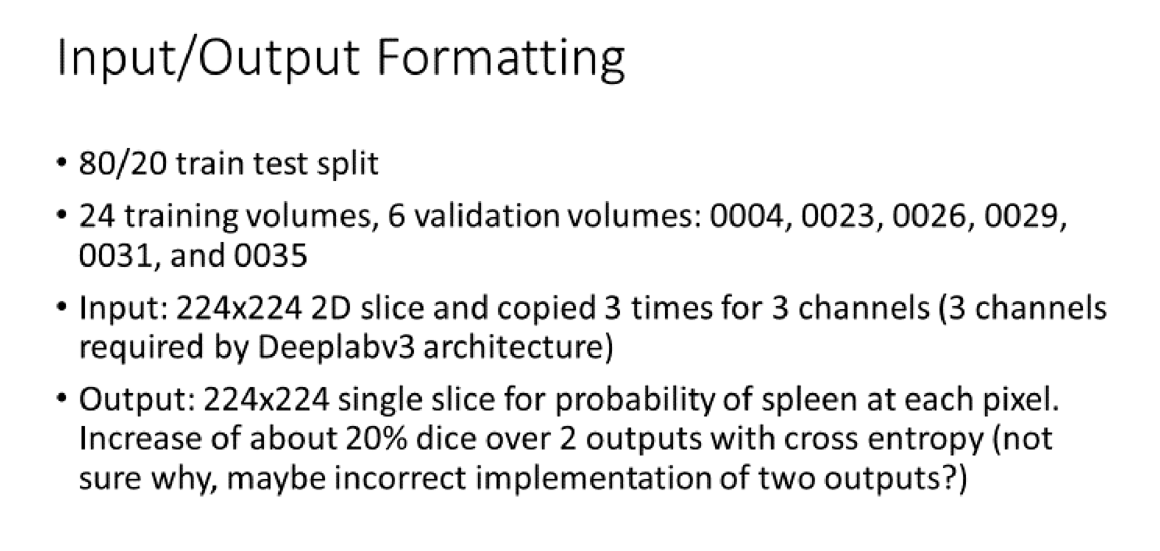


Architecture: deeplabv3

(b) Going deeper with atrous convolution. Atrous convolution with $rate > 1$ is applied after block3 when *output_stride* = 16.
Figure 3. Cascaded modules without and with atrous convolution.

Chen, Liang-Chieh, et al. "Rethinking atrous convolution for semantic image segmentation." *arXiv preprint arXiv:1706.05587* (2017).

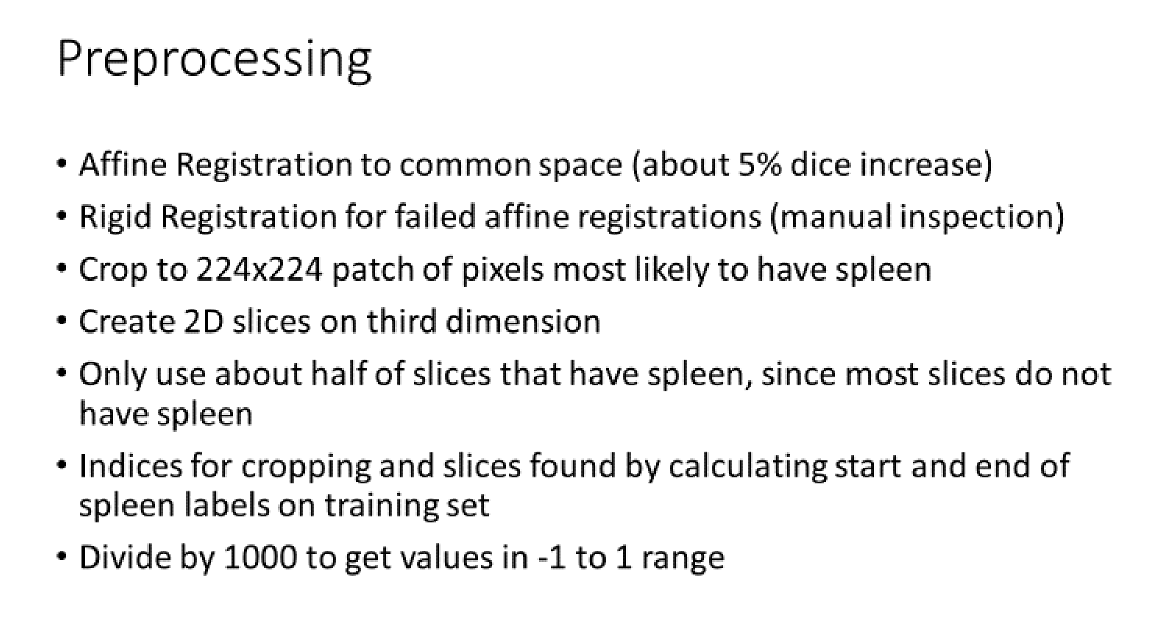
Slide 5



Input/Output Formatting

- 80/20 train test split
- 24 training volumes, 6 validation volumes: 0004, 0023, 0026, 0029, 0031, and 0035
- Input: 224x224 2D slice and copied 3 times for 3 channels (3 channels required by Deeplabv3 architecture)
- Output: 224x224 single slice for probability of spleen at each pixel. Increase of about 20% dice over 2 outputs with cross entropy (not sure why, maybe incorrect implementation of two outputs?)

Slide 6



Preprocessing

- Affine Registration to common space (about 5% dice increase)
- Rigid Registration for failed affine registrations (manual inspection)
- Crop to 224x224 patch of pixels most likely to have spleen
- Create 2D slices on third dimension
- Only use about half of slices that have spleen, since most slices do not have spleen
- Indices for cropping and slices found by calculating start and end of spleen labels on training set
- Divide by 1000 to get values in -1 to 1 range

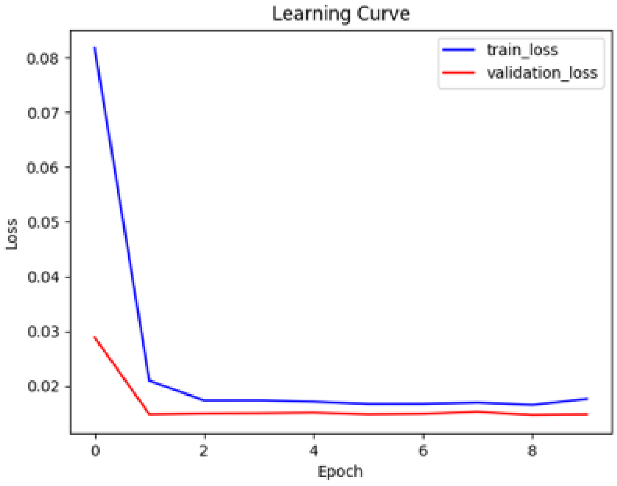
Slide 7

Postprocessing

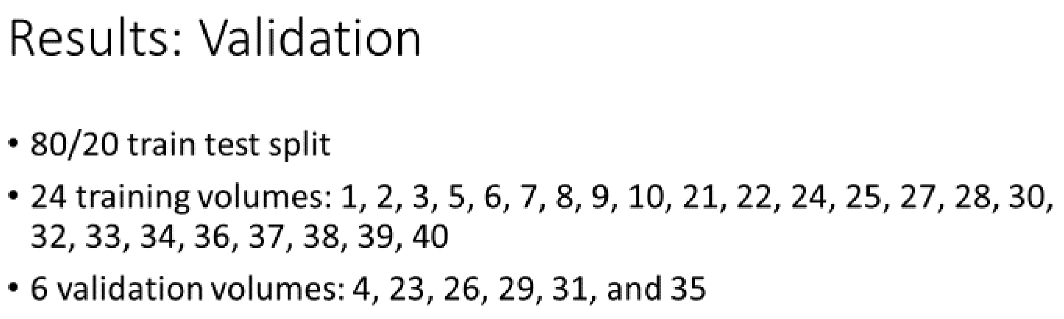
- Find optimal threshold for positive class using validation set (final threshold at 0)
- Register thresholded predictions back to original image space

Slide 8

Results: Learning Curve



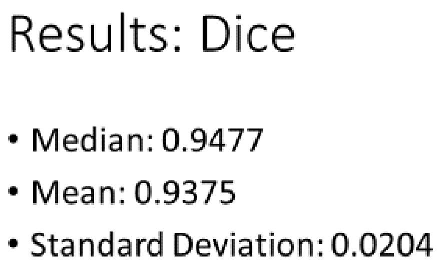
Slide 9



Results: Validation

- 80/20 train test split
- 24 training volumes: 1, 2, 3, 5, 6, 7, 8, 9, 10, 21, 22, 24, 25, 27, 28, 30, 32, 33, 34, 36, 37, 38, 39, 40
- 6 validation volumes: 4, 23, 26, 29, 31, and 35

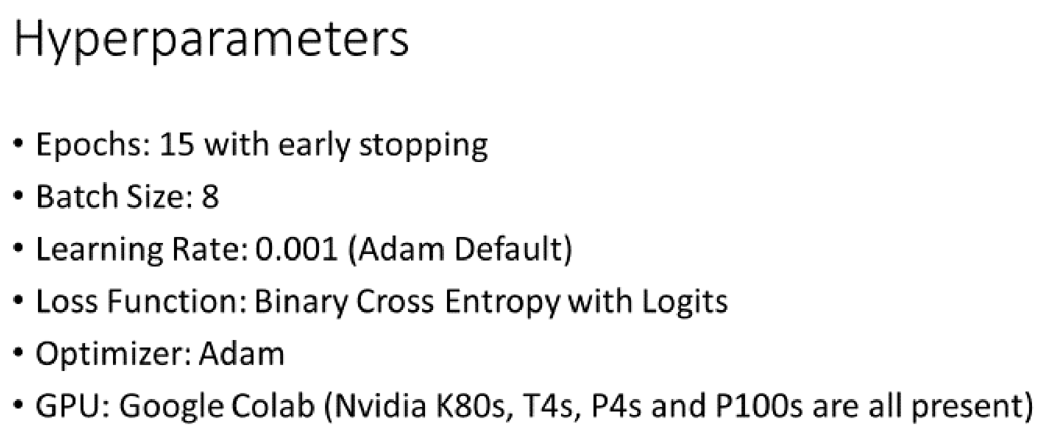
Slide 10



Results: Dice

- Median: 0.9477
- Mean: 0.9375
- Standard Deviation: 0.0204

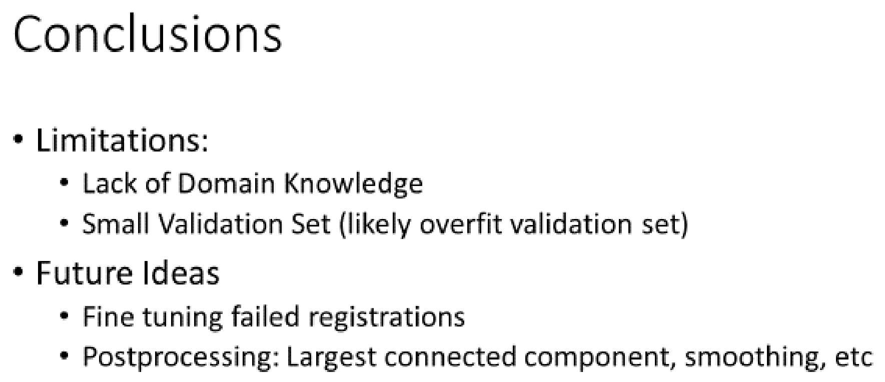
Slide 11



Hyperparameters

- Epochs: 15 with early stopping
- Batch Size: 8
- Learning Rate: 0.001 (Adam Default)
- Loss Function: Binary Cross Entropy with Logits
- Optimizer: Adam
- GPU: Google Colab (Nvidia K80s, T4s, P4s and P100s are all present)

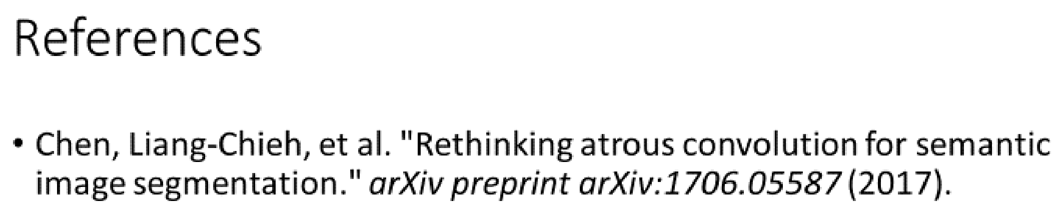
Slide 12



Conclusions

- Limitations:
  - Lack of Domain Knowledge
  - Small Validation Set (likely overfit validation set)
- Future Ideas
  - Fine tuning failed registrations
  - Postprocessing: Largest connected component, smoothing, etc

Slide 13



References

- Chen, Liang-Chieh, et al. "Rethinking atrous convolution for semantic image segmentation." *arXiv preprint arXiv:1706.05587* (2017).

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)
```