# A1_part1_diabetic_retinothapy

October 12, 2021

## 1 BIODS220 Assignment 1 Part a: Diabetic Retinopathy

### 1.1 Marc Huo

Diabetic retinopathy (DR) is an eye disease prevalent in diabetic patients. It is the leading cause of blindness in people aged 20-64. Screening for DR allows earlier and more effective treatment options, and accurate screening can save the eyesight of millions. A deep-learning approach to predicting DR from eye images was proposed in 2016, in the paper "Development and Validation of a Deep Learning Algorithm for Detection of Diabetic Retinopathy in Retinal Fundus Photographs", published in JAMA.

In this assignment we will use a similar dataset of retina images hosted by Kaggle. We'll train a model to predict whether or not to refer a patient for treatment given binarized severity of DR in patients: no referral if [No DR, mild] and referral if [moderate, severe, and proliferate DR].

### 1.2 Section 1: Data

Let's first explore the dataset, which is hosted by Kaggle. Go to https://www.kaggle.com/c/diabetic-retinopathy-detection/data, create a Kaggle account, then accept the competition rules after clicking the "Late Submission" button.

In your working environment run `pip install kaggle`.

Go to the 'Account' tab of your user profile and select 'Create New API Token'. Your personal `kaggle.json` file, which contains your Kaggle username and key, will download. Copy in your username and key to the following lines and run it in your Terminal to set these variables in your environment.

`export KAGGLE_USERNAME=your_user_name`

`export KAGGLE_KEY=your_key`

Downloading this dataset takes a while, so we recommend you use `screen` or `tmux` and do other tasks. Run the following in the directory where you want to put the data.

`kaggle competitions download -c diabetic-retinopathy-detection` (about 25 - 30 min)

`unzip diabetic-retinopathy-detection.zip "train*"` (~25 min)

`rm diabetic-retinopathy-detection.zip` # Do this **after** running the unzip command.

`cat train.zip.* > train.zip` (~20 min)

`unzip train.zip` (~15 min)

`unzip trainLabels.csv.zip` (~1 min)

You'll now have a directory called `train/` with the images we'll use for this notebook.

**Q1a.1**: Read over the data descriptions from Kaggle. Write down the source of data and list the the two ways that images are shown.

*Written answer*: Retinal images were provided by EyePACS, which is a free platform for retinopathy screening. The images provided are high resolution of two different types: one of the left field and one of the right field.

Import the necessary libraries and set our path by running the cell below. Replace IMAGE_PATH with the path to the `train` directory with all the images. Replace LABEL_PATH with the path to `trainLabels.csv`.

[2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random
import os
import cv2
from tensorflow.keras.preprocessing.image import ImageDataGenerator,␣
 ↪array_to_img, img_to_array, load_img
import tensorflow as tf
from sklearn.metrics import roc_curve

random.seed(2) # don't change this

# FILL IN CODE HERE #

IMAGE_PATH = '/home/marchuo/assign1/data/train/' # replace with your path
LABEL_PATH = '/home/marchuo/assign1/data/trainLabels.csv' # replace with your␣
 ↪path

# FILL IN CODE HERE #
```

Before beginning the data processing, let's first analyze the composition of our dataset.

**Q1a.2**: There are five classes: No DR (0), mild (1), moderate (2), severe (3), and proliferate DR (4). Graph a histogram for the classes. Write down the percentage of the largest class in the written answer below.

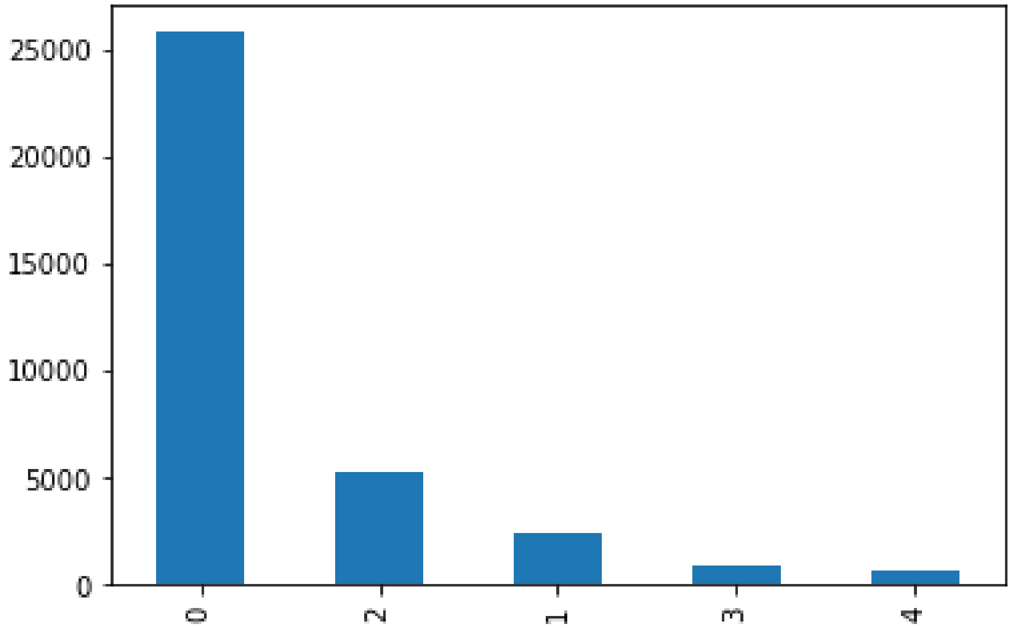
Hints: use a pandas DataFrame to read the labels data. To create the histogram, you can either use pandas, or a combination of numpy and matplotlib.

*Written answer*: 73.5%

[19]:
```
# FILL IN CODE HERE #
labels = pd.read_csv(LABEL_PATH)
labels["level"].value_counts().plot.bar()
```

```
zero_labels = labels[labels["level"] == 0].value_counts(normalize=True) * 100
print(zero_labels.shape[0] / labels.shape[0])

# FILL IN CODE HERE #
```

```
0.7347833513636622
```



**Q1a.3**: As you can see from the histogram, our dataset is very imbalanced. Imbalanced datasets are common in healthcare, especially in ophthalmology and pathology.

Give one reason why imbalanced datasets are common in health care, and suggest a problem that imbalanced data could cause.

*Written answer*: Imbalanced datasets are common in health care, because we have less data and records for people with rarer ailments such as diabetic retinopathy in comparison to the general population. Thus, due to the skew of data in which minority classes have only a few data entries in comparison to the majority class, there poses a challenge for our machine learning models. Since most models assume equal class size, this results in poor predictive performance of the minority class due to a bias towards a class. The best differentiation threshold shifts due to the data imbalance.

We want to use a deep learning model to refer patients to doctors based on the severity of risk seen in fundus photos. Therefore we will explore binary classification of 'no refer' and 'refer'. We will need binary labels, bucketing: - [No DR, mild] are 'no refer' - [Moderate, Severe, Proliferate] are 'refer'.

Because our dataset is imbalanced, we'll take 2000 random samples from 'no refer' calss and 2000 random samples from 'refer' class. (This is the crudest method for dealing with imbalanced data, but it will be good enough to get decent results in this exercise).

**Q1a.4**: DR levels [0, 1] will be 'no refer' and DR levels [2, 3, 4] will be 'refer'. Randomly sample 2000 from each type by creating 2 lists of image filenames: `no_refer_examples` and `refer_examples`. For example, refer_examples will consist of ['11503_right', '44093_right', etc], if these image names are DR levels 2, 3, or 4.

[3]:
```
# FILL IN CODE HERE #
refer = (labels[labels.level >= 2].sample(2000))
refer_examples = refer["image"].to_numpy()

no_refer = (labels[labels.level <= 1].sample(2000))
no_refer_examples = no_refer["image"].to_numpy()
# FILL IN CODE HERE #
```

**Q1a.5**: Complete the `load_data` function to generate a train/validation/test sets with split `(0.6,0.2,0.2)`.

Read in the images and labels to numpy arrays using Keras functions `load_img` and `img_to_array`. (If you get a dependency error for `Pillow`, then just run `pip install Pillow` in the VM terminal.) For each image: - resize to `IMAGE_SIZE` using `cv2` with interpolation flag `INTER_LANCZOS4` - normalize the pixels in each image with the max and min pixel value of that image (so each image will have min pixel value 0, and max pixel value 1.

The expected shapes of the output are: - `X_train` is`(2400, 224, 224, 3)`; `X_val` and `X_test` is `(800, 224, 224, 3)` - `y_train` is `(2400,)`; `y_val` and `y_test` is `(800,)`

[4]:
```
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split

IMAGE_SIZE = (224, 224)

def normalize_resize_img(image_path):
    img = load_img(image_path)
    img_array = img_to_array(img)
    resized_img = cv2.resize(img_array, IMAGE_SIZE, interpolation=cv2.
 ↪INTER_LANCZOS4)
    norm_img = cv2.normalize(resized_img, None, alpha=0, beta=1, norm_type=cv2.
 ↪NORM_MINMAX)
    return norm_img

def load_data(IMAGE_PATH, LABEL_PATH, split=(0.6,0.2,0.2)):
    """
    Load batches of images and labels. Splits images and labels
    into arrays

    Parameters:
    IMAGE_PATH (str/path): path to directory with images.
    LABEL_PATH (str/path): path to directory with labels.
    split (tuple): 3 values summing to 1 defining split of train, validation␣
 ↪and test
```

```
    Returns:
    X_train (np.ndarray): Train images. A numpy array of shape (N_train, 224,␣
↪224, 3)
    y_train (np.ndarray): Train labels. A numpy array of shape (N_train,)
    X_val (np.ndarray): Val images. A numpy array of shape (N_val, 224, 224, 3)
    y_val (np.ndarray): Val labels. A numpy array of shape (N_val,)
    X_test (np.ndarray): Test images. A numpy array of shape (N_test, 224, 224,␣
↪3)
    y_test (np.ndarray): Test labels. A numpy array of shape (N_test,)
    """
    train_images = []
    train_labels = []
    # FILL IN CODE HERE #
    for image in refer_examples:
        image_path = os.path.join(IMAGE_PATH, image + ".jpeg")
        resized_norm_img = normalize_resize_img(image_path)
        train_images.append(resized_norm_img)
        train_labels.append(1)

    for image in no_refer_examples:
        image_path = os.path.join(IMAGE_PATH, image + ".jpeg")
        resized_norm_img = normalize_resize_img(image_path)
        train_images.append(resized_norm_img)
        train_labels.append(0)

    # Shuffle train_images and train_labels while maintaining order
    train_images, train_labels = shuffle(train_images, train_labels)
    # FILL IN CODE HERE #

    all_images = np.stack(train_images)
    all_labels = np.array(train_labels).flatten()

    # FILL IN CODE HERE #
    # Split train/val/test by creating the returned variables #
    X_train, X_rem, y_train, y_rem = train_test_split(all_images, all_labels,␣
↪train_size=split[0])
    X_val, X_test, y_val, y_test = train_test_split(X_rem, y_rem,␣
↪test_size=split[1]/(split[1] + split[2]))
    # FILL IN CODE HERE #

    return X_train, y_train, X_val, y_val, X_test, y_test

X_train, y_train, X_val, y_val, X_test, y_test = load_data(IMAGE_PATH,␣
↪LABEL_PATH)

print(f"X_train shape {X_train.shape}")
print(f"y_train shape {y_train.shape}")
print(f"X_val shape {X_val.shape}")
print(f"y_val shape {y_val.shape}")
print(f"X_test shape {X_test.shape}")
print(f"y_test shape {y_test.shape}")
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
/tmp/ipykernel_1920/3011200973.py in <module>
     59     return X_train, y_train, X_val, y_val, X_test, y_test
     60 
---> 61 X_train, y_train, X_val, y_val, X_test, y_test = load_data(IMAGE_PATH,␣
 ↪LABEL_PATH)
     62 
     63 print(f"X_train shape {X_train.shape}")

/tmp/ipykernel_1920/3011200973.py in load_data(IMAGE_PATH, LABEL_PATH, split)
     34     for image in refer_examples:
     35         image_path = os.path.join(IMAGE_PATH, image + ".jpeg")
---> 36         resized_norm_img = normalize_resize_img(image_path)
     37         train_images.append(resized_norm_img)
     38         train_labels.append(1)

/tmp/ipykernel_1920/3011200973.py in normalize_resize_img(image_path)
      6 def normalize_resize_img(image_path):
      7     img = load_img(image_path)
----> 8     img_array = img_to_array(img)
      9     resized_img = cv2.resize(img_array, IMAGE_SIZE, interpolation=cv2.
 ↪INTER_LANCZOS4)
     10     norm_img = cv2.normalize(resized_img, None, alpha=0, beta=1,␣
 ↪norm_type=cv2.NORM_MINMAX)

/opt/conda/lib/python3.7/site-packages/keras/preprocessing/image.py in␣
 ↪img_to_array(img, data_format, dtype)
    241       dtype = backend.floatx()
    242     kwargs['dtype'] = dtype
--> 243   return image.img_to_array(img, data_format=data_format, **kwargs)
    244 
    245 

/opt/conda/lib/python3.7/site-packages/keras_preprocessing/image/utils.py in␣
 ↪img_to_array(img, data_format, dtype)
    307     # or (channel, height, width)
    308     # but original PIL image has format (width, height, channel)
--> 309     x = np.asarray(img, dtype=dtype)
    310     if len(x.shape) == 3:
    311         if data_format == 'channels_first':
```

```
/opt/conda/lib/python3.7/site-packages/numpy/core/_asarray.py in asarray(a,␣
↪dtype, order)
     81
     82     """
---> 83     return array(a, dtype, copy=False, order=order)
     84
     85

/opt/conda/lib/python3.7/site-packages/PIL/Image.py in __array__(self, dtype)
    701                 __array_interface__ = new
    702
--> 703         return np.array(ArrayData(), dtype)
    704
    705     def __getstate__(self):

KeyboardInterrupt:
```

This processing should take around 15 minutes to finish.

We recommend saving the data to a file using the next cell. Once you've done that, comment out that cell. When you need to load this data again in the future, ybcommenting out the code in the cell after and run it.

[5]:
```
"""
# # This method saves your data
saved_data_path='/home/marchuo/assign1/data/train/saved_data3' # change to your␣
↪path
with open(saved_data_path, 'wb') as f:
    np.save(f, X_train)
    np.save(f, y_train)
    np.save(f, X_val)
    np.save(f, y_val)
    np.save(f, X_test)
    np.save(f, y_test)
"""
```

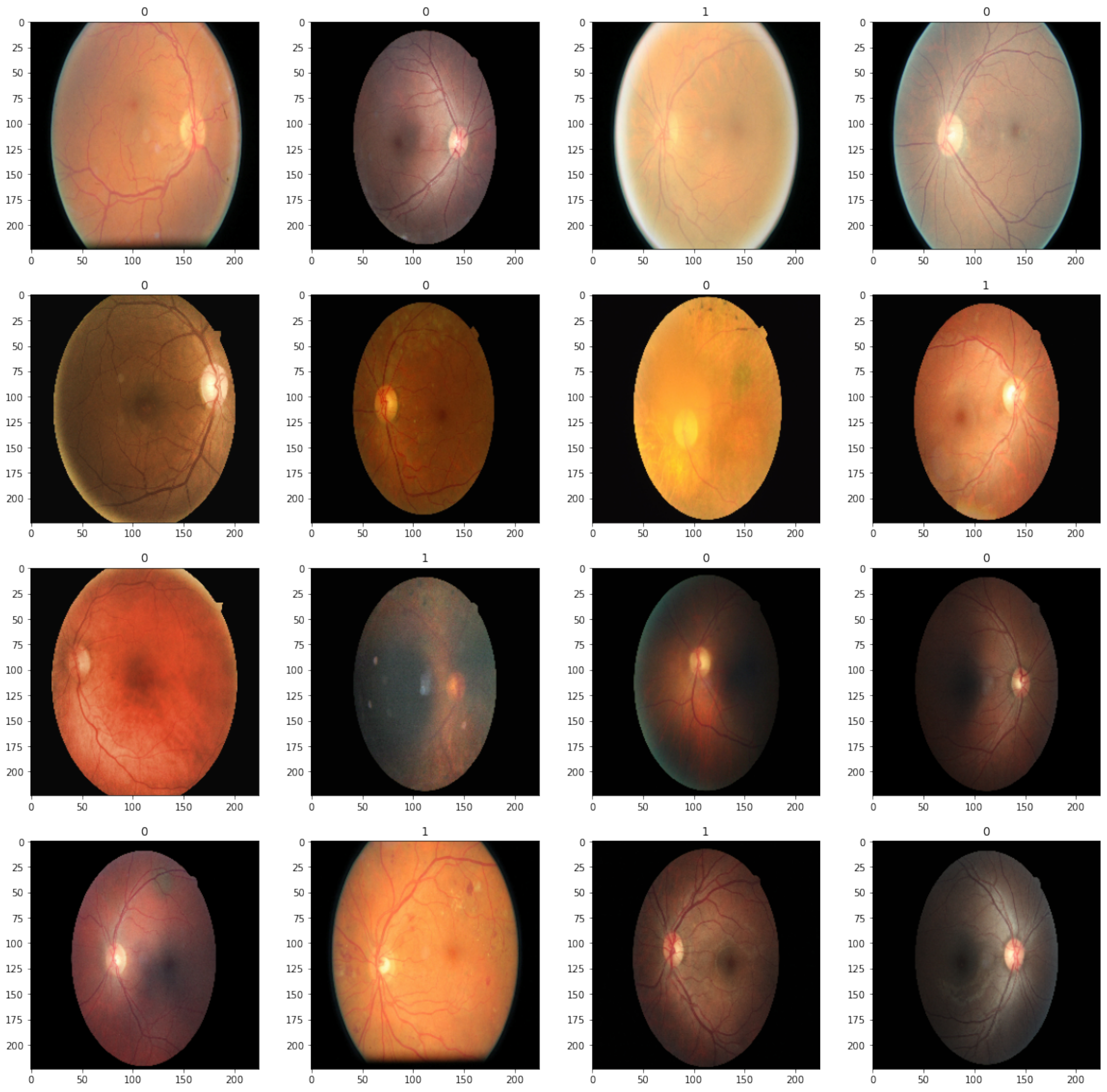
[5]:
```
"\n# # This method saves your
data\nsaved_data_path='/home/marchuo/assign1/data/train/saved_data3' # change to
your path\nwith open(saved_data_path, 'wb') as f:\n    np.save(f, X_train)\n
np.save(f, y_train)\n    np.save(f, X_val)\n    np.save(f, y_val)\n
np.save(f, X_test)\n    np.save(f, y_test)\n"
```

[6]:
```
# This method loads your data
saved_data_path='/home/marchuo/assign1/data/train/saved_data3'
with open(saved_data_path, 'rb') as f:
    X_train = np.load(f)
    y_train = np.load(f)
    X_val = np.load(f)
    y_val = np.load(f)
    X_test = np.load(f)
    y_test = np.load(f)
```

Let's sanity check our load data function by printing out the first five train examples with the label of each example as the title.

[7]:
```
nrows, ncols = 4,4
f, axs = plt.subplots(nrows, ncols, figsize=(20,20))
for i in range(nrows):
    for j in range(ncols):
        indx = i*nrows+j
        axs[i,j].imshow(X_train[indx])
        axs[i,j].set(title=y_train[indx])
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



**Q1a.6**: We've split the input into train, validation, and test sets. We will now train the model with the train set, and use performance on the validation set to tune hyperparameters. Finally, we will evaluate on the test set.

Explain the purpose of the test set, and why we don't use it till the end of the analysis.

*Written answer*: The purpose of the test set is to assess the performance and generalization of the machine learning model fit and predictive power. It is only used once the model has been trained with the train and validation sets. We don't see the test set until the end of the analysis, because we want to prevent the model from training on the test data - this test set is used as a "real world" set of what kind of data the model will be used to predict on.

Run the below cells to create train, validation, and test datasets from our preprocessed data.

```
[8]: BATCH_SIZE = 10

train_dset = tf.data.Dataset.from_tensor_slices((X_train, y_train)).
 ↪batch(BATCH_SIZE)
val_dset = tf.data.Dataset.from_tensor_slices((X_val, y_val)).batch(BATCH_SIZE)
test_dset = tf.data.Dataset.from_tensor_slices((X_test, y_test)).
 ↪batch(BATCH_SIZE)
```

```
2021-10-12 02:07:41.436968: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2021-10-12 02:07:41.542695: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2021-10-12 02:07:41.543531: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2021-10-12 02:07:41.552915: I tensorflow/core/platform/cpu_feature_guard.cc:142]
This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN) to use the following CPU instructions in performance-critical
operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
2021-10-12 02:07:41.553675: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2021-10-12 02:07:41.554550: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2021-10-12 02:07:41.555314: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2021-10-12 02:07:43.564253: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2021-10-12 02:07:43.565136: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2021-10-12 02:07:43.565958: I
```

```
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2021-10-12 02:07:43.567644: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1510] Created device
/job:localhost/replica:0/task:0/device:GPU:0 with 10819 MB memory:  -> device:
0, name: Tesla K80, pci bus id: 0000:00:04.0, compute capability: 3.7
2021-10-12 02:07:43.587480: W
tensorflow/core/framework/cpu_allocator_impl.cc:80] Allocation of 1445068800
exceeds 10% of free system memory.
2021-10-12 02:07:44.504527: W
tensorflow/core/framework/cpu_allocator_impl.cc:80] Allocation of 1445068800
exceeds 10% of free system memory.
2021-10-12 02:07:45.031306: W
tensorflow/core/framework/cpu_allocator_impl.cc:80] Allocation of 481689600
exceeds 10% of free system memory.
2021-10-12 02:07:45.295286: W
tensorflow/core/framework/cpu_allocator_impl.cc:80] Allocation of 481689600
exceeds 10% of free system memory.
2021-10-12 02:07:45.463415: W
tensorflow/core/framework/cpu_allocator_impl.cc:80] Allocation of 481689600
exceeds 10% of free system memory.
```

### 1.3 Section 2: Model

**Q1a.7**: Let's create a model for DR using keras. We will be using a MobileNet (https://arxiv.org/abs/1704.04861) pre-trained on ImageNet as our base model and fine-tuning it on our dataset. This is called transfer learning. MobileNet is a light weight convolutional neural network structure. It is efficient and also achieves comparable accuracy to that of Inception and ResNet for most problems, while requiring less memory and compute to train.

First, specify `tf.keras.applications.mobilenet_v2.MobileNetV2` as our base_model (see the documentation for Keras saved models https://keras.io/api/applications/mobilenet/). Set `weights="imagenet"` and `include_top=False` because we want to remove the last layer (which has 1000 nodes corresponding to the 1000 ImageNet classes; we want to have only one output node for our binary classification task).

Use `tf.keras.Sequential` to add the following layers: - `average_layer` by using a global averaging layer from the Keras api. This aggregates the model outputs from the base layer. - One dense `prediction_layer`, so that our final layer is the right dimension for binary classification - An `activation` layer that will force the input into the appropriate range for binary classification.

Compile your model using the Keras commands and include: - An appropriate loss function for binary classification. - Use the `Adam` optimizer with initial learning rating of 0.1. - Add the `accuracy` metric.

[9]:
```
# FILL IN CODE HERE #
base = tf.keras.applications.mobilenet_v2.MobileNetV2(weights="imagenet",␣
 ↪include_top=False)
```

```
model = tf.keras.Sequential(
    layers=[
        base,
        tf.keras.layers.GlobalAveragePooling2D(),
        tf.keras.layers.Dense(1, activation='sigmoid')
    ]
)

model.compile(loss=tf.keras.losses.BinaryCrossentropy(), optimizer=tf.keras.
 ↪optimizers.Adam(learning_rate=0.0001), metrics=['accuracy'])
# FILL IN CODE HERE #
```

```
WARNING:tensorflow:`input_shape` is undefined or non-square, or `rows` is not in
[96, 128, 160, 192, 224]. Weights for input shape (224, 224) will be loaded as
the default.
```

**Q1a.8**: Call `model.fit` (https://keras.io/models/model/) with 30 epochs on the train set and validate on the validation set. Don't worry, you should get poor performance for these current starter hyperparameters. This could be due to learning rate, choice of optimizer, or choice of loss function.

Tune your learning rate, and rerun `model.fit` to achieve better accuracy on the tune set. You should be able to get above 0.9 accuracy for train and above 0.6 accuracy for validation. Report your accuracies below.

[10]:
```
hist = model.fit(
# FILL IN CODE HERE #
    train_dset, epochs=30, validation_data=val_dset
# FILL IN CODE HERE #
)
```

```
Epoch 1/30

2021-10-12 02:07:53.722042: I
tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR
Optimization Passes are enabled (registered 2)
2021-10-12 02:07:55.418252: I tensorflow/stream_executor/cuda/cuda_dnn.cc:369]
Loaded cuDNN version 8005

240/240 [==============================] - 50s 142ms/step - loss: 0.6711 -
accuracy: 0.6137 - val_loss: 1.0026 - val_accuracy: 0.5562
Epoch 2/30
240/240 [==============================] - 33s 138ms/step - loss: 0.3448 -
accuracy: 0.8600 - val_loss: 2.4390 - val_accuracy: 0.5412
Epoch 3/30
240/240 [==============================] - 33s 138ms/step - loss: 0.1288 -
accuracy: 0.9638 - val_loss: 4.1802 - val_accuracy: 0.5425
Epoch 4/30
240/240 [==============================] - 33s 138ms/step - loss: 0.1610 -
accuracy: 0.9396 - val_loss: 4.9719 - val_accuracy: 0.5437
```

```
Epoch 5/30
240/240 [==============================] - 33s 138ms/step - loss: 0.1630 -
accuracy: 0.9429 - val_loss: 6.1675 - val_accuracy: 0.5525
Epoch 6/30
240/240 [==============================] - 33s 138ms/step - loss: 0.1128 -
accuracy: 0.9642 - val_loss: 5.9173 - val_accuracy: 0.5562
Epoch 7/30
240/240 [==============================] - 33s 137ms/step - loss: 0.0646 -
accuracy: 0.9771 - val_loss: 5.8815 - val_accuracy: 0.5587
Epoch 8/30
240/240 [==============================] - 33s 137ms/step - loss: 0.0558 -
accuracy: 0.9812 - val_loss: 3.8833 - val_accuracy: 0.6062
Epoch 9/30
240/240 [==============================] - 33s 137ms/step - loss: 0.0510 -
accuracy: 0.9862 - val_loss: 2.7965 - val_accuracy: 0.6438
Epoch 10/30
240/240 [==============================] - 33s 137ms/step - loss: 0.0363 -
accuracy: 0.9883 - val_loss: 2.4946 - val_accuracy: 0.6425
Epoch 11/30
240/240 [==============================] - 33s 138ms/step - loss: 0.0309 -
accuracy: 0.9892 - val_loss: 1.3743 - val_accuracy: 0.6787
Epoch 12/30
240/240 [==============================] - 33s 137ms/step - loss: 0.0327 -
accuracy: 0.9892 - val_loss: 1.7999 - val_accuracy: 0.7063
Epoch 13/30
240/240 [==============================] - 33s 137ms/step - loss: 0.0467 -
accuracy: 0.9862 - val_loss: 2.6193 - val_accuracy: 0.6700
Epoch 14/30
240/240 [==============================] - 34s 140ms/step - loss: 0.0790 -
accuracy: 0.9700 - val_loss: 5.7449 - val_accuracy: 0.5788
Epoch 15/30
240/240 [==============================] - 33s 138ms/step - loss: 0.1106 -
accuracy: 0.9621 - val_loss: 5.1003 - val_accuracy: 0.6137
Epoch 16/30
240/240 [==============================] - 33s 137ms/step - loss: 0.0610 -
accuracy: 0.9775 - val_loss: 3.0388 - val_accuracy: 0.6762
Epoch 17/30
240/240 [==============================] - 33s 137ms/step - loss: 0.0131 -
accuracy: 0.9967 - val_loss: 3.3954 - val_accuracy: 0.6675
Epoch 18/30
240/240 [==============================] - 33s 138ms/step - loss: 0.0020 -
accuracy: 1.0000 - val_loss: 3.1624 - val_accuracy: 0.6825
Epoch 19/30
240/240 [==============================] - 33s 138ms/step - loss: 6.5499e-04 -
accuracy: 1.0000 - val_loss: 2.5159 - val_accuracy: 0.6925
Epoch 20/30
240/240 [==============================] - 33s 137ms/step - loss: 3.5759e-04 -
accuracy: 1.0000 - val_loss: 2.1278 - val_accuracy: 0.7088
```

```
Epoch 21/30
240/240 [==============================] - 33s 137ms/step - loss: 2.6573e-04 -
accuracy: 1.0000 - val_loss: 1.8880 - val_accuracy: 0.7175
Epoch 22/30
240/240 [==============================] - 33s 138ms/step - loss: 2.1140e-04 -
accuracy: 1.0000 - val_loss: 1.7347 - val_accuracy: 0.7250
Epoch 23/30
240/240 [==============================] - 33s 137ms/step - loss: 1.7348e-04 -
accuracy: 1.0000 - val_loss: 1.6364 - val_accuracy: 0.7200
Epoch 24/30
240/240 [==============================] - 33s 137ms/step - loss: 1.4509e-04 -
accuracy: 1.0000 - val_loss: 1.5707 - val_accuracy: 0.7250
Epoch 25/30
240/240 [==============================] - 33s 137ms/step - loss: 1.2297e-04 -
accuracy: 1.0000 - val_loss: 1.5274 - val_accuracy: 0.7250
Epoch 26/30
240/240 [==============================] - 33s 137ms/step - loss: 1.0524e-04 -
accuracy: 1.0000 - val_loss: 1.4998 - val_accuracy: 0.7225
Epoch 27/30
240/240 [==============================] - 33s 137ms/step - loss: 9.0722e-05 -
accuracy: 1.0000 - val_loss: 1.4829 - val_accuracy: 0.7237
Epoch 28/30
240/240 [==============================] - 33s 137ms/step - loss: 7.8655e-05 -
accuracy: 1.0000 - val_loss: 1.4739 - val_accuracy: 0.7225
Epoch 29/30
240/240 [==============================] - 33s 137ms/step - loss: 6.8500e-05 -
accuracy: 1.0000 - val_loss: 1.4703 - val_accuracy: 0.7175
Epoch 30/30
240/240 [==============================] - 33s 136ms/step - loss: 5.9887e-05 -
accuracy: 1.0000 - val_loss: 1.4706 - val_accuracy: 0.7150
```

*Accuracy for tune*: 1.00 test accuracy, 0.71 validation accuracy

*Accuracy for test*: 0.4992 test accuracy, 0.53 validation accuracy

**Q1a.9**: Tuning hyperparameters is crucial to deep learning! Explain the changes you made to tune the model, and suggest why it may have improved training.

*Written answer*: Learning rate is a hyperparameter that controls the rate at which a model algorithm updates parameter estimates and weights, effectively controlling the rate at which the model learns. To improve the model accuracy, I changed the learning rate or step size of the model to 0.001. As a result, it allows the model to learn a more optimal set of weights for each parameter albeit at a slower rate. This was able to remedy the previous issue of the model having too large of a step size which resulted in weight updates that were too large.

## 1.4 Section 3: Evaluation

Evaluation is one of the most important parts of medical classification, as it helps us determine what to do with the model predictions and how it can improve screening processes.

We've used the train and test set to create the model. Notice that (`X_test`, and `y_test`) were not used in that process. It would be very bad practice to evaluate the model on the test set, and then return and update the model based on those results (then the test set is acting like just another validation set).

We'll explore different ways to assess our model in the next questions.

**Q1a.10**: The simplest evaluation is accuracy. Use `model.evaluate` (https://keras.io/models/model/) with `X_test` and `y_test` and print the test accuracy (this will work if you compiled the model with the accuracy metric earlier). You should have an accuracy of above 0.6.

[14]:
```
# FILL IN CODE HERE #
model.evaluate(X_test, y_test)
y_pred = model.predict(X_test).ravel()
# FILL IN CODE HERE #
```

```
25/25 [==============================] - 2s 83ms/step - loss: 1.7117 - accuracy:
0.6900
```

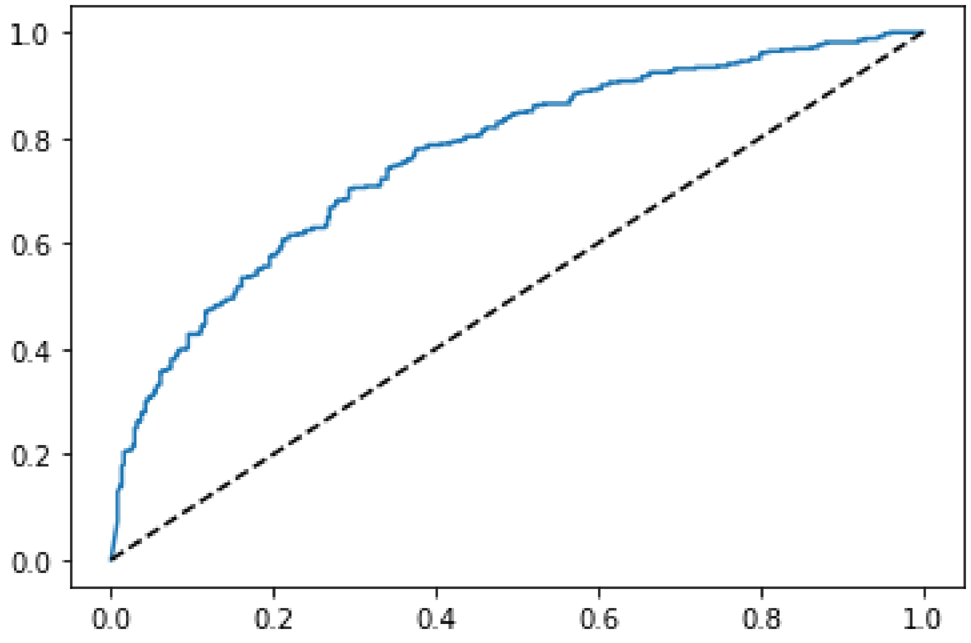
**Q1a.11**: Let's go deeper by considering the balance of false-positive and false-negative rates. Use `sklearn.metrics.roc_curve` to compute false-positive and true-positive rates at different levels of binary threshold. Plot the ROC curve, and then use `sklearn.metrics.roc_auc_score` to compute the AUC (area under the curve of the ROC curve).

In the written answer below, interpret what the AUC means.

[15]:
```
from sklearn.metrics import roc_curve, roc_auc_score
def check_binarized_auc(test_dset, model):
    """
    Given a trained model and test dataset, compute the parameters of the ROC␣
 ↪curve
    and compute the AUC.

    Parameters:
    test_dset (tf.data.Dataset): test dataset
    model (tf.keras.model): prediction model

    Returns:
    fpr (np.ndarray): same as docstring of sklearn.metrics.roc_curve
    tpr (np.ndarray): same as docstring of sklearn.metrics.roc_curve
    thresholds (np.ndarray): same as docstring of sklearn.metrics.roc_curve
    auc (float): area under the ROC curve for the curve defined by fpr, tpr,␣
 ↪and thresholds.
    """
    # FILL IN CODE HERE #
    fpr, tpr, thresholds = roc_curve(y_test, y_pred)
    auc = roc_auc_score(y_test, y_pred)
    # FILL IN CODE HERE #
    return fpr, tpr, thresholds, auc
```

```
fpr, tpr, thresholds, auc = check_binarized_auc(test_dset, model)
plt.plot(fpr, tpr)
plt.plot([0, 1], [0, 1], 'k--')
print(f"AUC score={auc}")
```

```
AUC score=0.7663340833171428
```



*Written answer*: The AUC score is the measure of the ability of a model to distinguish between two classes - in this case, the model has an AUC score of 0.76, measuring its ability to distinguish between the positive and negative classes. In this model, it measures the ability of the model to classify between refer and no refer examples. An AUC of 0.76 adequately discriminates between classes, but still is not optimal - it is better than the AUC of randomly discriminating (0.5), but worse than the AUC of a perfect discriminating model (1.0).

**Q1a.12**: Finally, let's choose an operating point using Youden's index (https://en.wikipedia.org/wiki/Youden%27s_J_statistic) and report its sensitivity and specificity at that point.

[17]:
```
def youdensj(fpr,tpr,thresholds):
    j = tpr-fpr
    max_j = np.argmax(j)
    best_threshold = thresholds[max_j]
    return best_threshold

def choose_operating_point(fpr, tpr, thresholds):
    """
    Given ROC curve parameters, choose an operating point
    using the Youden J statistic

    Parameters:
    fpr, tpr, thresholds (np.ndarray): same as docstring for␣
↪`check_binarized_auc`

    Returns:
    op_point (float): operating point threshold for binary task chosen with␣
↪Youdens J statistic
    sens (float): test dataset sensitivity score, https://en.wikipedia.org/wiki/
↪Sensitivity_and_specificity
    spec (float): test dataset specificity score, https://en.wikipedia.org/wiki/
↪Sensitivity_and_specificity
    """

    # FILL IN CODE HERE #
    op_point = youdensj(fpr, tpr, thresholds)
    i = np.where(thresholds == op_point)
    sens = tpr[i][0]
    spec = (1 - fpr)[i][0]
    # FILL IN CODE HERE #
    return op_point, sens, spec

op_point, sens, spec = choose_operating_point(fpr, tpr, thresholds)
print(f'Operating point: {op_point}')
print(f'Sensitivity: {sens}')
print(f'Specificity: {spec}')
```

```
Operating point: 0.3087770640850067
Sensitivity: 0.7030878859857482
Specificity: 0.7071240105540897
```

**Q1a.13**: What does it mean to choose an operating point with high sensitivity, and an operating point with high specificity? List one medical application where high sensitivity is needed, and one when high specificity is needed.

*Written answer*: An operating point with high sensitivity corresponds to high negative predictive value, which is ideal for "rule-out" tests. On the other hand, an operating point with high specificity corresponds to high positive predictive value, which is ideal for "rule-in" tests. An example medical application of high sensitivity corresponds to cases where we want to predict whether a patient has a disease or screening applications. For high specificity, it can be used to test negativity in health, where we want to know the proportion of population without the disease and give negative test results. High specificity should be used to make decisions about high-risk actions.

# A1_part2_2d_lung_segmentation

October 12, 2021

## 1 BIODS220 Assignment 1 Part b: 2D Lung Segmentation

According to the World Heath Organization, lung cancer is the leading cause of cancer-related deaths worldwide, accounting for an estimated 1.4 million deaths in 2018 (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3864624/). CT scans are now being used in the United States for screening high risk individuals for lung cancer. In order to detect cancerous lesions in scans, a useful first step is to segment the lungs in the image. (Then a researcher would use just the segmented image as input to another model). In this notebook we will perform segmentation on lungs in CT scans using a U-Net model (https://sites.pitt.edu/~sjh95/related_papers/u-net.pdf).

[1]:
```
%env TF_CPP_MIN_LOG_LEVEL=3  # silence some TensorFlow warnings and logs.

import os
import cv2
import numpy as np
import tensorflow as tf
import pandas as pd
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.image import ImageDataGenerator,␣
↪array_to_img, img_to_array, load_img

# Please feel free to play with hyperparameters, but submit the assignment with␣
↪the original values.
IMAGE_SIZE = (128, 128)
TRAIN_VAL_TEST_SPLIT = (0.6, 0.2, 0.2)
BATCH_SIZE = 16
SEED = 220
EPOCHS = 40
```

```
env: TF_CPP_MIN_LOG_LEVEL=3  # silence some TensorFlow warnings and logs.
```

### 1.1 Section 1: Data

We will be using a Kaggle dataset from https://www.kaggle.com/kmader/finding-lungs-in-ct-data. Similar to part 1, create an API token to set in your environment and download the dataset from Kaggle through the following commands.

```
kaggle datasets download -d kmader/finding-lungs-in-ct-data
```

```
unzip finding-lungs-in-ct-data.zip
```

This should take a couple of minutes. Fill in the IMAGE_DIR and MASK_DIR with the output paths.

[2]:
```
# FILL IN CODE HERE #

IMAGE_DIR = '/home/marchuo/assign1/data/lung/2d_images' # replace with your path
MASK_DIR = '/home/marchuo/assign1/data/lung/2d_masks' # replace with your path
PROJECT_DIR = '/home/marchuo/assign1/'

# FILL IN CODE HERE #
```

**Q1b.1**: First let's load the dataset from the saved folders. Our dataset contains 267 CT scans and the corresponding segmentations. Implement the `load_data` function below.

For each file (CT scan or mask):

1) Normalize the image with its maximum and minmum value, so each image will be in the range [0,1].
2) Read the image using `cv2.imread`. Make sure to read all channels with the `cv2.IMREAD_UNCHANGED` argument.
3) For the segmentation mask only, convert the numpy array to type `int16`.
4) Resize the image to `IMAGE_SIZE` using the `cv2.resize` function. For the image use `cv2.INTER_LANCZOS4` interpolator. For the segmentation mask use `INTER_NEAREST`.

Then split all the data into three training (60% of data), validation (20% of data), and test (20% of data). The expected shapes of the `train_images` and `train_masks` arrays are (160, 128, 128, 1).

[3]:
```
import random

def load_data(IMAGE_DIR, MASK_DIR):
    """ Load images and mask from the saved folders, IMG_DIR and MASK_dir and␣
 ↪split them to
        train, validation, and test sets.

        Input:
            IMAGE_DIR (str): path to the folder containing ct scan images
            MASK_DIR (str): path to the folder containing corresponding masks
        Output:
            train_images, val_images, test_images (numpy array of shape (number␣
 ↪of images, IMAGE_SIZE)):
                preprocessed images split in 0.6, 0.2, 0.2 respectively
            train_masks, val_masks, test_masks (numpy array of shape (number of␣
 ↪masks, IMAGE_SIZE)):
                preprocessed masks split in 0.6, 0.2, 0.2 respectively
    """
    images = []
    masks = []
    for i, file in enumerate(sorted(os.listdir(IMAGE_DIR))):
        image_path = os.path.join(IMAGE_DIR, file)
        mask_path = os.path.join(MASK_DIR, file)

        # FILL IN CODE HERE #
        image = cv2.imread(image_path, cv2.IMREAD_UNCHANGED)
        resized_img = cv2.resize(image, IMAGE_SIZE, interpolation=cv2.
 ↪INTER_LANCZOS4)
        norm_image = cv2.normalize(resized_img, None, alpha=0, beta=1,␣
 ↪norm_type=cv2.NORM_MINMAX)
        images.append(norm_image)

        mask = cv2.imread(mask_path, cv2.IMREAD_UNCHANGED).astype(np.int16)
        resized_mask = cv2.resize(mask, IMAGE_SIZE, cv2.INTER_NEAREST)
        norm_mask = cv2.normalize(resized_mask, None, alpha=0, beta=1,␣
 ↪norm_type=cv2.NORM_MINMAX)
        masks.append(norm_mask)
        # FILL IN CODE HERE #

    all_images = np.stack(images)[:,:,:,np.newaxis]
    all_masks = np.stack(masks)[:,:,:,np.newaxis]

    # FILL IN CODE HERE #
    train_images, remaining_images = all_images[:round(0.6 * all_images.
 ↪shape[0]), :, :, :], all_images[round(0.6 * all_images.shape[0]):, :, :, :]
    test_images, val_images = np.array_split(remaining_images, 2)

    train_masks, remaining_masks = all_masks[:round(0.6 * all_masks.shape[0]), :
 ↪, :, :], all_masks[round(0.6 * all_masks.shape[0]):, :, :, :]
    test_masks, val_masks = np.array_split(remaining_masks, 2)
    # FILL IN CODE HERE #

    return train_images, train_masks, val_images, val_masks, test_images,␣
 ↪test_masks

train_images, train_masks, val_images, val_masks, test_images, test_masks =␣
 ↪load_data(IMAGE_DIR, MASK_DIR)
print('Train images: ', train_images.shape)
print('Train mask: ', train_masks.shape)
```

```
Train images:  (160, 128, 128, 1)
Train mask:  (160, 128, 128, 1)
```

**Q1b.2**: We will now implement data augmentation. As we can see from the previous part, we only have 160 training examples. Neural networks usually need large number of training examples to succeed in learning a task such as segmentation. Therefore, we will use data augmentation techniques to simulate exposing our model to many more training examples. We will use the following augmentations on our data:

1) Random horizontal shift with at most 0.1 of image width
2) Random vertical shift with at most 0.1 of image height
3) Random rotation with at most 10 degrees
4) Random zoom with at most 0.1 times zoom-in or zoom-out

Explain why these augmentations should help improve the accuracy of the model. Is it possible for data augmentation to instead decrease the performance of the trained model on the test set? If so, explain under what situation this could happen.

*Written answer*: Data augmentation is useful to help improve performance of machine learning models, because it introduces new input data to train datasets. The horizontal/vertical shift, random rotation, and random zoom introduces variability and size to our dataset by shifting the pixels horizontally/vertically while keeping dimensions the same, or randomly rotates the pixels, or adds new pixel values around the image or interpolates pixel values, respectively.

Some augmentation may lead to decreased accuracy if the model does not have enough capacity. Additionally, depending on the augmentation method, it may lead to adding an invalid augmented image for training purposes (e.g. flipped photo) which may harm training. Augmentation may also result in vital image features being cut off from the image. Underfitting may also be a potential result of data augmentation.

**Q1b.3**: For data augmentation in Keras, we will use the `ImageDataGenerator` class which has several built-in data augmentation methods. Look at the documentation of ImageDataGenerator and implement the following function. You only need to define `image_datagen` and `mask_datagen`.

The final print statement will check that your data generator is returning data with the right shape

```
[4]: from tensorflow.keras.preprocessing.image import ImageDataGenerator

     def create_generator(train_images, train_masks):
         """ Creat a generator based on training images and masks.

             Input:
                 train_images (np): array of training images
                 train_masks (np): array of training masks
             Output:
                 data_generator (generator function): generator which yields images␣
      ↪and masks after data augmenetation
         """
         # FILL IN CODE HERE #
         image_datagen = ImageDataGenerator(width_shift_range=0.1,
             height_shift_range=0.1, zoom_range=0.1, rotation_range = 10
         )
         mask_datagen = ImageDataGenerator(width_shift_range=0.1,
             height_shift_range=0.1, zoom_range=0.1, rotation_range = 10
```

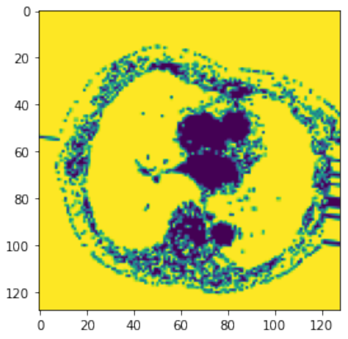
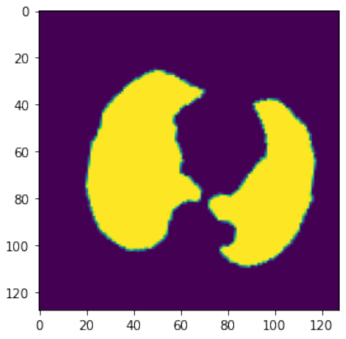
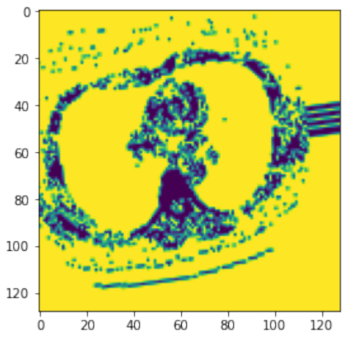
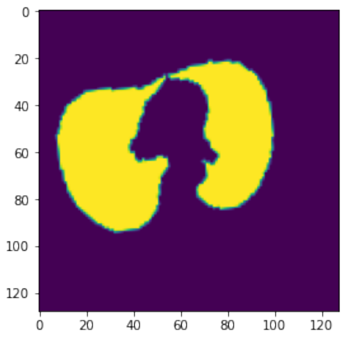
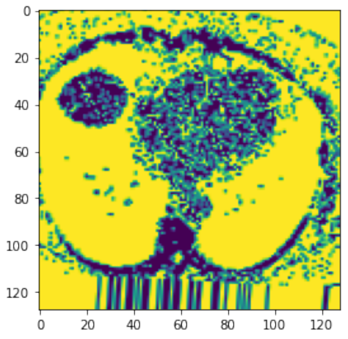
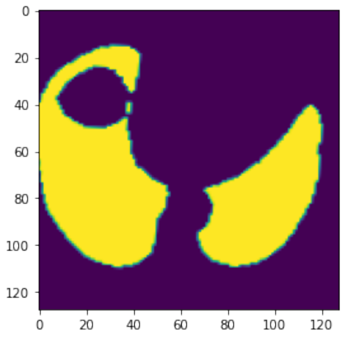
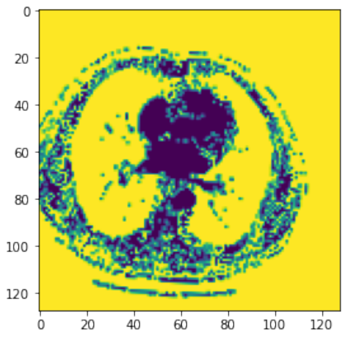
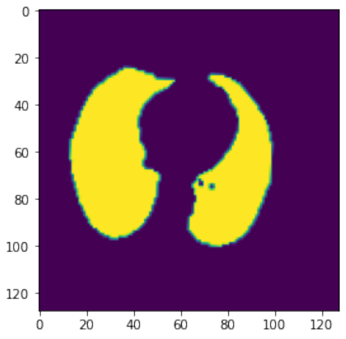
```
    )
    # FILL IN CODE HERE #

    image_datagen.fit(train_images, augment=True, seed=SEED)
    mask_datagen.fit(train_masks, augment=True, seed=SEED)

    image_generator = image_datagen.flow(train_images, batch_size=BATCH_SIZE,␣
 ↪seed=SEED)
    mask_generator = mask_datagen.flow(train_masks, batch_size=BATCH_SIZE,␣
 ↪seed=SEED)
    data_generator = zip(image_generator, mask_generator)


    return data_generator

data_generator = create_generator(train_images, train_masks)
#print(f"Image batch shape {img.shape}, mask batch shape {mask.shape}")
```

Run the following code to convert the generator (for the training set) and numpy arrays (for the validation and test sets) to the TensorFlow dataset format. Don't worry about these details; in part 3 of this assignment you will get practice dealing with generators for tensorflow datasets.

[5]:
```
def train_generator():
    while True:
        image_batch, mask_batch = next(data_generator)
        yield image_batch, mask_batch

train_dataset = tf.data.Dataset.from_generator(train_generator,
                     output_types=(tf.float32, tf.float32),
                     output_shapes=([None, IMAGE_SIZE[0], IMAGE_SIZE[1], 1],
                                    [None, IMAGE_SIZE[0], IMAGE_SIZE[1], 1])
                                             ).repeat()
val_dataset = tf.data.Dataset.from_tensor_slices((val_images.astype(np.
 ↪float32), val_masks)).batch(BATCH_SIZE)
test_dataset = tf.data.Dataset.from_tensor_slices((test_images.astype(np.
 ↪float32), test_masks)).batch(BATCH_SIZE)
```

**Q1b.4**: Let's take a look at some image examples and their corresponding masks from the training dataset. Take a batch from the `train_dataset` and plot four images and their corresponding mask.

[6]:
```
# FILL IN CODE HERE #
for images, masks in train_dataset.take(1):
    fig, ax = plt.subplots(figsize=(20, 20), nrows=4, ncols=2)
    for i in range(4):
        ax[i,0].imshow(images[i].numpy())
        ax[i,1].imshow(masks[i].numpy())
# FILL IN CODE HERE #
```

## 1.2 Section 2: Model

Let's implement the U-Net model for our segmentation task. You can see the model architecture described in the U-Net paper here: https://sites.pitt.edu/~sjh95/related_papers/u-net.pdf, figure 1. We'll be implementing a simpler version of this without a weighted loss for class imbalance.

**Q1b.5**: At a high level, explain how U-Net works and why it is more suitable for the segmentation task relative to a simple convolutional neural network (CNN).

*Written answer*: U-Net consists of a contracting and expansive path, explaining its U shape. The contracting path consists of the same elements as a typical CNN, with a repeated application of convolutions, a ReLU, and a max pooling operation. Pooling operations are replaced by upsampling operators, hence increasing the resolution of the output. There are a lot feature channels on upsampling, which allows the network to include context information to high resolution layers. The expansive path is relatively symmetric to the contracting path, which yields the u-shaped architecture and recreates the image vector.

The U-Net is more suitable for the segmentation task relative to a simple convolutional neural network. In the case of a CNN, the image is converted into a vector to be used for classification. However, it does not reconstruct the image from the vector - on the other hand, the U-Net architecture turns the feature mapping it learned from image to vector conversion, and it reapplies it to expand the vector to the segmented image. Thus, structural integrity of the image is preserved and distortion is reduced.

**Q1b.6**: Implement a simplified version of U-Net based on Figure 1 of the paper using the Keras Model API with the constraints: - Use only these layers: `Conv2D`, `MaxPool2D`, `UpSampling2D`, and `Dropout`. You will also use the `concat` operation. - The architecture (the order of `Conv2D`, `MaxPool2D` and `UpSampling2D` layers) should be similar to the U-Net architecture. - Model accuracy on the validation set must be at least 90%, and it will be possible to achieve higher than 95%.

You can choose any architecture that meets those constraints, and you should get decent results as long as long as you approximately follow the UNet structure. But if you'd like more direction, here are guidelines for an architecture that achieved the desired accuracy. We will contrast this suggested architecture with the more complicated architecture in Figure 1 of the paper: - **Max-Pool/Upsampling steps**: 2 `MaxPool2D` layers on the left branch, and 2 `UpSampling2D` layers on the right branch (original paper architecture has 4 each). - **Filters**: The Unet filter count increases after max pooling steps, and decreases back again as we upsample. Our suggested architecture uses filter counts of [64,128,256] on the down-branch, and [256,128,64] on the up-branch (original paper has [64,128,256,512,1024] because they downsample twice more). - **Conv Layers**: Only 1 `Conv2d` after each upsampling or downsampling step (original paper has 2 Conv2d layers). Choose `padding="same"` which keeps the same input and output shape (original paper uses `padding=None` and handles shape mismatch with cropping). Use `ReLU` activations (same as original paper). - **Up-conv layer**: the `up-conv` layer is defined as `UpSampling2D` followed by a `Conv2D`. The recommended architecture uses this structure (which is the same as original paper), though you can get okay results by only using `UpSampling2D`. - **Dropout**: one dropout layer before the final `Conv2D` layer. - **Final layer**: A `Conv2d` layer that has an output with 1 channel restricted to the range [0,1].

[7]:
```
from tensorflow.keras.layers import Conv2D, MaxPool2D, UpSampling2D, Dropout,␣
 ↪concatenate
class U_Net(tf.keras.Model):
    def __init__(self):
        super(U_Net, self).__init__()

        # FILL IN CODE HERE #
        # First conv layer
        self.conv1 = Conv2D(64, (3, 3), padding="same", activation='relu')␣
↪#concat

        self.pool1 = MaxPool2D(pool_size=(2, 2))

        # Second conv layer
        self.conv2 = Conv2D(128, (3, 3), padding="same", activation='relu')␣
↪#concat

        self.pool2 = MaxPool2D(pool_size=(2, 2))

        # Third conv layer
        self.conv3 = Conv2D(256, (3, 3), padding="same", activation='relu')

        self.upsample1 = UpSampling2D((2, 2))


        self.upconv1 = Conv2D(256, (3, 3), padding="same", activation='relu')

        self.upsample2 = UpSampling2D((2, 2))

        self.upconv2 = Conv2D(128, (2, 2), padding="same", activation='relu')

        self.upconv3 = Conv2D(64, (2, 2), padding="same", activation='relu')

        self.dropout = Dropout(0.5)

        self.output_layer = Conv2D(1, (1, 1), activation='sigmoid')

        # FILL IN CODE HERE #

    def call(self, inputs):
        # FILL IN CODE HERE #

        conv1 = self.conv1(inputs)

        output = self.pool1(conv1)

        conv2 = self.conv2(output)

        output = self.pool2(conv2)

        output = self.conv3(output)

        upsample1 = self.upsample1(output)
        output = concatenate([upsample1, conv2], axis=-1)

        output = self.upconv1(output)

        upsample2 = self.upsample2(output)

        output = concatenate([upsample2, conv1], axis=-1)

        output = self.upconv3(output)

        output = self.dropout(output)

        output = self.output_layer(output)

        # FILL IN CODE HERE #

        return output

# create model
model = U_Net()

# test that the output data shape is reasonable
img_batch, mask_batch = next(iter(test_dataset))
sample_out = model(img_batch)
print(f"Input shape {img_batch.shape}, model out shape {sample_out.shape}")
```

```
Input shape (16, 128, 128, 1), model out shape (16, 128, 128, 1)
```

**Q1b.7**: Which loss function do you think is most appropriate for our task here? Why? After selecting the loss function, compile the model using the Adam optimizer with default learning rate. Add the accuracy metric.

(Note that the UNet paper describes a special weighting coefficient that makes its loss function different to the one you may choose; you don't have to implement this weighting.)

*Written answer*: Binary cross entropy is the most appropriate for our task here, because we are attempting to do pixel-wise binary classification. Every output is independent and not mutually exclusive, while also taking on values of 0 or 1. In this case, our labels are our masks, where our "0-class" pixels are 0 and our "1-class" pixels are 1.

[8]:
```
# FILL IN CODE HERE #
model.compile('adam', 'binary_crossentropy', 'accuracy'

)
# FILL IN CODE HERE #
```

**Q1b.8**: Before training the model, let's first set a learning rate scheduler. This will adjust the learning rate during the training of the model. Why can it be useful to use a learning rate schedule to train complex neural networks?

*Written answer*: Learning rate scheduling can increase performance and decrease training time for complex neural networks. The most common and used learning rate schedule is one that decreases over iteration and time. The model starts off with a large learning rate, resulting in large changes to model weights at the beginning of the training procedure. Then, over time, learning rate decreases and smaller training updates are made to the weights. This is particularly effective, because it allows for large changes for the untrained model in the beginning and finetuning of weights as more epochs are iterated over.

Now we are going to train our model for 40 epochs. Implement the following function such that the learning rate for the first 20 epochs is 0.001, and after that it decreases exponentially with a factor of 0.1. For more information, please look at the `ExponentialDecay` documentation in Keras.

```
[9]: def scheduler(epoch):
         """ Schedule the learning rate to be 0.0001 for first
             20 epoch, and decrease exponentially by a factor of 0.1
             for all remaining epochs.

             Input:
                 epoch (int)
             Output:
                 learning_rate (float)
         """
         # FILL IN CODE HERE #
         initial_learning_rate = 0.001
         scheduler = tf.keras.optimizers.schedules.ExponentialDecay(
             initial_learning_rate, 20, 0.9, staircase=True)
         return scheduler
         # FILL IN CODE HERE #

     lr_scheduler = tf.keras.callbacks.LearningRateScheduler(scheduler)
```

**Q1b.9**: Train the model using the `model.fit` function in the Keras API. We compute the value of `steps_per_epoch` which tells the API how many times to sample from the tensorflow dataset object. It should be passed as an argument to `model.fit()`.

```
[10]: steps_per_epoch = int(len(os.listdir(IMAGE_DIR)) * TRAIN_VAL_TEST_SPLIT[0]) //␣
       ↪BATCH_SIZE # pass this arg to fit()


      hist = model.fit(
          # FILL IN CODE HERE #
          train_dataset, epochs = 40, validation_data=val_dataset,␣
       ↪steps_per_epoch=steps_per_epoch
          # FILL IN CODE HERE #
      )
```

```
Epoch 1/40
10/10 [==============================] - 10s 552ms/step - loss: 0.5613 -
```

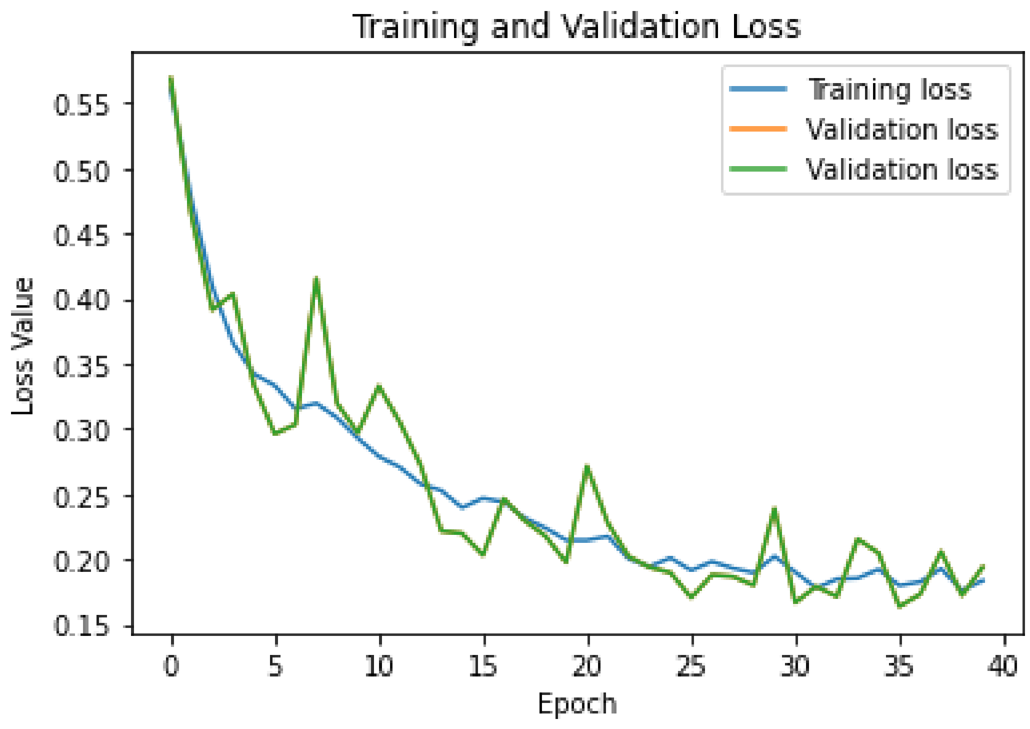
```
accuracy: 0.7081 - val_loss: 0.5688 - val_accuracy: 0.7584
Epoch 2/40
10/10 [==============================] - 4s 404ms/step - loss: 0.4761 -
accuracy: 0.7461 - val_loss: 0.4636 - val_accuracy: 0.7584
Epoch 3/40
10/10 [==============================] - 4s 401ms/step - loss: 0.4096 -
accuracy: 0.7388 - val_loss: 0.3915 - val_accuracy: 0.7584
Epoch 4/40
10/10 [==============================] - 4s 401ms/step - loss: 0.3659 -
accuracy: 0.7426 - val_loss: 0.4038 - val_accuracy: 0.7584
Epoch 5/40
10/10 [==============================] - 4s 401ms/step - loss: 0.3423 -
accuracy: 0.7906 - val_loss: 0.3339 - val_accuracy: 0.8634
Epoch 6/40
10/10 [==============================] - 4s 402ms/step - loss: 0.3334 -
accuracy: 0.8419 - val_loss: 0.2963 - val_accuracy: 0.8886
Epoch 7/40
10/10 [==============================] - 4s 400ms/step - loss: 0.3155 -
accuracy: 0.8633 - val_loss: 0.3035 - val_accuracy: 0.8877
Epoch 8/40
10/10 [==============================] - 4s 398ms/step - loss: 0.3198 -
accuracy: 0.8464 - val_loss: 0.4154 - val_accuracy: 0.8669
Epoch 9/40
10/10 [==============================] - 4s 402ms/step - loss: 0.3086 -
accuracy: 0.8637 - val_loss: 0.3196 - val_accuracy: 0.8927
Epoch 10/40
10/10 [==============================] - 4s 399ms/step - loss: 0.2929 -
accuracy: 0.8722 - val_loss: 0.2971 - val_accuracy: 0.8949
Epoch 11/40
10/10 [==============================] - 4s 402ms/step - loss: 0.2790 -
accuracy: 0.8774 - val_loss: 0.3330 - val_accuracy: 0.8915
Epoch 12/40
10/10 [==============================] - 4s 400ms/step - loss: 0.2709 -
accuracy: 0.8805 - val_loss: 0.3053 - val_accuracy: 0.8951
Epoch 13/40
10/10 [==============================] - 4s 399ms/step - loss: 0.2579 -
accuracy: 0.8836 - val_loss: 0.2726 - val_accuracy: 0.9065
Epoch 14/40
10/10 [==============================] - 4s 400ms/step - loss: 0.2529 -
accuracy: 0.8871 - val_loss: 0.2219 - val_accuracy: 0.9178
Epoch 15/40
10/10 [==============================] - 4s 400ms/step - loss: 0.2397 -
accuracy: 0.8864 - val_loss: 0.2200 - val_accuracy: 0.9185
Epoch 16/40
10/10 [==============================] - 4s 398ms/step - loss: 0.2470 -
accuracy: 0.8855 - val_loss: 0.2033 - val_accuracy: 0.9222
Epoch 17/40
10/10 [==============================] - 4s 400ms/step - loss: 0.2441 -
```

```
accuracy: 0.8856 - val_loss: 0.2466 - val_accuracy: 0.9109
Epoch 18/40
10/10 [==============================] - 4s 398ms/step - loss: 0.2319 -
accuracy: 0.8936 - val_loss: 0.2298 - val_accuracy: 0.9153
Epoch 19/40
10/10 [==============================] - 4s 402ms/step - loss: 0.2242 -
accuracy: 0.8971 - val_loss: 0.2180 - val_accuracy: 0.9194
Epoch 20/40
10/10 [==============================] - 4s 399ms/step - loss: 0.2148 -
accuracy: 0.9030 - val_loss: 0.1977 - val_accuracy: 0.9267
Epoch 21/40
10/10 [==============================] - 4s 399ms/step - loss: 0.2147 -
accuracy: 0.9012 - val_loss: 0.2715 - val_accuracy: 0.9047
Epoch 22/40
10/10 [==============================] - 4s 399ms/step - loss: 0.2178 -
accuracy: 0.9034 - val_loss: 0.2279 - val_accuracy: 0.9201
Epoch 23/40
10/10 [==============================] - 4s 405ms/step - loss: 0.2005 -
accuracy: 0.9106 - val_loss: 0.2023 - val_accuracy: 0.9265
Epoch 24/40
10/10 [==============================] - 4s 397ms/step - loss: 0.1945 -
accuracy: 0.9125 - val_loss: 0.1940 - val_accuracy: 0.9308
Epoch 25/40
10/10 [==============================] - 4s 399ms/step - loss: 0.2012 -
accuracy: 0.9105 - val_loss: 0.1899 - val_accuracy: 0.9287
Epoch 26/40
10/10 [==============================] - 4s 398ms/step - loss: 0.1916 -
accuracy: 0.9150 - val_loss: 0.1707 - val_accuracy: 0.9376
Epoch 27/40
10/10 [==============================] - 4s 401ms/step - loss: 0.1984 -
accuracy: 0.9138 - val_loss: 0.1881 - val_accuracy: 0.9316
Epoch 28/40
10/10 [==============================] - 4s 399ms/step - loss: 0.1933 -
accuracy: 0.9138 - val_loss: 0.1869 - val_accuracy: 0.9319
Epoch 29/40
10/10 [==============================] - 4s 399ms/step - loss: 0.1898 -
accuracy: 0.9152 - val_loss: 0.1801 - val_accuracy: 0.9355
Epoch 30/40
10/10 [==============================] - 4s 397ms/step - loss: 0.2023 -
accuracy: 0.9099 - val_loss: 0.2389 - val_accuracy: 0.9149
Epoch 31/40
10/10 [==============================] - 4s 398ms/step - loss: 0.1903 -
accuracy: 0.9155 - val_loss: 0.1667 - val_accuracy: 0.9387
Epoch 32/40
10/10 [==============================] - 4s 399ms/step - loss: 0.1780 -
accuracy: 0.9208 - val_loss: 0.1793 - val_accuracy: 0.9364
Epoch 33/40
10/10 [==============================] - 4s 399ms/step - loss: 0.1850 -
```

```
accuracy: 0.9174 - val_loss: 0.1714 - val_accuracy: 0.9390
Epoch 34/40
10/10 [==============================] - 4s 398ms/step - loss: 0.1856 -
accuracy: 0.9175 - val_loss: 0.2157 - val_accuracy: 0.9189
Epoch 35/40
10/10 [==============================] - 4s 400ms/step - loss: 0.1924 -
accuracy: 0.9138 - val_loss: 0.2049 - val_accuracy: 0.9276
Epoch 36/40
10/10 [==============================] - 4s 400ms/step - loss: 0.1800 -
accuracy: 0.9188 - val_loss: 0.1638 - val_accuracy: 0.9364
Epoch 37/40
10/10 [==============================] - 4s 400ms/step - loss: 0.1828 -
accuracy: 0.9183 - val_loss: 0.1733 - val_accuracy: 0.9380
Epoch 38/40
10/10 [==============================] - 4s 396ms/step - loss: 0.1929 -
accuracy: 0.9149 - val_loss: 0.2060 - val_accuracy: 0.9276
Epoch 39/40
10/10 [==============================] - 4s 399ms/step - loss: 0.1761 -
accuracy: 0.9208 - val_loss: 0.1723 - val_accuracy: 0.9393
Epoch 40/40
10/10 [==============================] - 4s 397ms/step - loss: 0.1838 -
accuracy: 0.9199 - val_loss: 0.1941 - val_accuracy: 0.9291
```

Using the `hist` variable returned from running the `fit` command above, run the following code to plot the training and validation loss.

[11]:
```
plt.figure()
plt.plot(range(EPOCHS), hist.history['loss'], label='Training loss')
plt.plot(range(EPOCHS), hist.history['val_loss'], label='Validation loss')
plt.plot(range(EPOCHS), hist.history['val_loss'], label='Validation loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss Value')
plt.legend()
plt.show()
```



### 1.3 Section 3: Evaluation

**Q1b.10**: Use `model.evaluate` to evaluate the accuracy of the model on the test set. You should have an accuracy greater than 0.9.

[12]:
```
model.evaluate(test_dataset)
```

```
4/4 [==============================] - 2s 539ms/step - loss: 0.1497 - accuracy:
0.9423
```

[12]:
```
[0.14971326291561127, 0.9422878623008728]
```

**Q1b.11**: Based on the loss and accuracy on the test set, do you think we are overfitting? Please describe three methods that can be used to avoid overfitting.

*Written answer*: Based on the loss and accuracy on the test set, I don't believe we are overfitting the data.

1) A dropout layer can be used for regularization to prevent neural networks from overfitting. It randomly drops neurons from the neural network during training in each iteration, resulting in an equivalent effect of training different neural networks.
2) Early stopping could be used as a form of regularization to prevent neural networks from overfitting. The idea of early stopping is to prevent models from fitting the training data past a certain point. Up to a point, the model better fits the training set and increases performance on the test set - however, after past that point, improving the model's fit to the

training data results in increase in generalization error. Early stopping stops the moel from training after a certain amount of iterations to prevent this issue.

3) Data augmentation can be used to prevent neural networks from overfitting. The idea is to increase the amount of images present in the dataset by flipping, translating, adding noise, rotating, and scaling original data. Thus, the dataset size increases and reduces overfitting - the model is unable to overfit all the samples and generalizes.

**Q1b.12**: What is Intersection over Union (IoU) and why it is a useful metric for the segmentation task? Implement the following function using the definition of IoU. Note that the `np.logical_and` and `np.logical_or` could be useful.

*Written answer*: The Intersection over Union (IoU) is a method to compute the percent overlap between the ground truth mask and our prediction output. This is a particularly useful metric for the segmentation task in order to evaluate how well our model is able to segment our region of interest by comparing it to the target mask and its overlap

[88]:
```
def compute_IoU(target, prediction):
    """
    Evaluate the intersection over union score for a single prediction and␣
 ↪ground truth value

    Parameters:
    target: (np.ndarray) : The ground truth label values
    prediction (np.ndarray): The labels predicted by the model
    """
    # FILL IN CODE HERE #
    intersection = np.logical_and(target, prediction)
    union = np.logical_or(target, prediction)
    iou = np.sum(intersection) / np.sum(union)
    # FILL IN CODE HERE #
    return iou
```

**Q1b.13**: Using the compute_IoU function, compute the mean IoU on the test set and print the result. Compute the IoU for each image separately, and then take the mean over images. You can binarize the prediction mask with a threshold of 0.1.

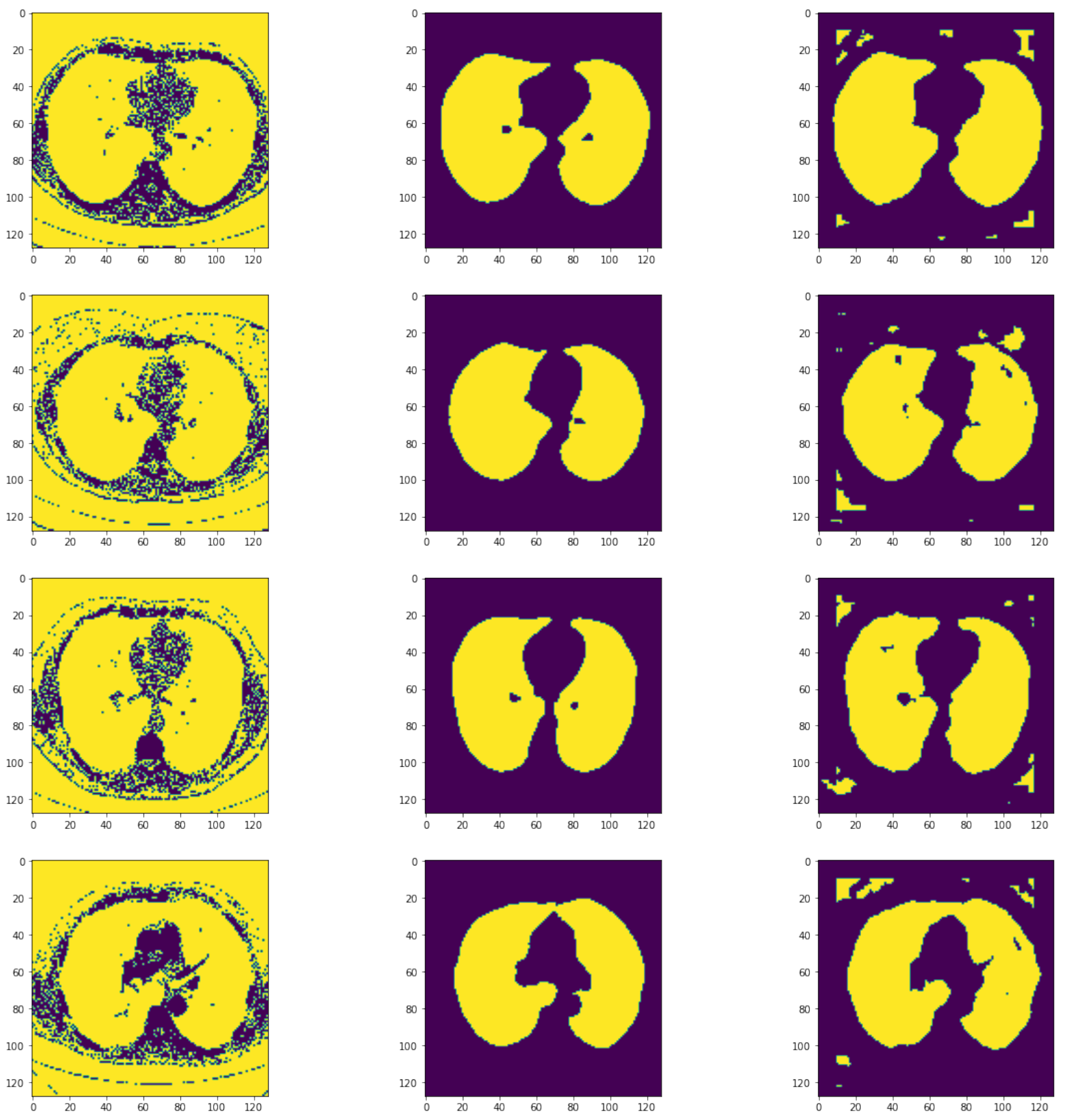
[117]:
```
# FILL IN CODE HERE #
pred = model.predict(test_dataset)
thresh = 0.1
iou_arr = []
selected_images = []
for n, mask in enumerate(test_masks):
    bin_img = np.where(pred[n] > thresh, 1, 0)
    iou = compute_IoU(mask, bin_img)
    iou_arr.append(iou)
    if iou > 0.85:
        selected_images.append((iou, n, mask, bin_img))

iou_mean = np.mean(iou_arr)
```

```
print(iou_mean)
# FILL IN CODE HERE #
```

```
0.6865233087397785
```

**Q1b.14**: Let's make some predictions! Plot four test images, their corresponding ground truth mask, and the model prediction. Make sure to show at least one image for each of the IoU ranges of [0.85, 0.9], [0.9, 0.95], and [0.95, 1].

[121]:
```
# FILL IN CODE HERE #
selected_images = sorted(selected_images, key=lambda x: x[0], reverse=True)
fig, ax = plt.subplots(figsize=(20, 20), nrows=4, ncols=3)
for n, (iou, index, true, pred) in enumerate(selected_images):
    ax[n, 0].imshow(test_images[index])
    ax[n, 1].imshow(true)
    ax[n, 2].imshow(pred)
    if n == 3:
        break
# FILL IN CODE HERE #
```



# A1_part3_3d_lung_segmentation

October 14, 2021

## 1 BIODS220 Assignment 1 Part c: 3D Lung Segmentation

Following our previous 2D lung segmentation project, we will now be extending the work to 3D lung segmentation using a 3D U-Net. This will allow us to learn more 3D based structures with 3D convolutions, especially as 3D voxelized input is commonly seen in biomedical datasets.

We will be using 3D CT scans from the same Kaggle challenge as the last part of the assignment. However, we'll be using a different part of the dataset with a much smaller number of complete 3D scans instead of 2D scans. Our goal is to conduct volumetric 3D segmentation on our test set.

But labelling a complete 3D scan is time intensive, and rarely will we have complete 3D scans at hand. In this assignment, we'll be trying to see whether training with sparsity in the train set can still allow us to approach the accuracy of training the full train set. Sparsity in our case refers to using a smaller percent of labels in our 3D volume, for example, having 30% of 2D slices labelled, while the other 70% is unlabelled. We'll be simulating this scenario below as a case of semi-supervised learning.

We will be using NiBabel to read in the data for this assignment. You can install it with `pip install nibabel`.

**Q1c.1**: Define semi-supervised learning and list out two examples of semi-supervised tasks with both input and output specified.

*Written answer*: Semi-supervised learning combines a small amount of labeled data with a large amount of unlabeled data. It is a learning model that sits somewhere between a supervised and unsupervised learning model, where the algorithm learns from a dataset that includes both labeled and unlabeled data. Semi-supervised learning is particularly useful when you don't have enough labeled data, and semi-supervised techniques can be used to increase your training dataset. Similarly, it is particularly beneficial when you have insufficient domain knowledge to label or insufficient time to prepare the data labels. Two semi-supervised learning methods/tasks are outlined below:

1) Semi-supervised learning for speech quality assessment is a common application as labeling speech data is time-intensive. The model is trained on the transformation of speech signals to the feature space model, which is accomplished by dividing speech signals to short frames with a short-term spectral analysis method. The dataset contains these short frames of speech signals as either labeled or unlabeled instances. With speech quality assessment, the output of the classification was which speaker the input signal mapped to.

2) Internet Content Classification is a common semi-supervised learning task as labeling all webpages of a dataset is extremely time-sensitive. A massive set of documents is to be classified into several classes - a very small subset of documents were labeled from each class

to train the semi-supervised model. With NLP, document classification was accomplished by factors including length of document, topic, word usage, and syntax. The output is classification of a document into separate classes.

[193]:
```
%env TF_CPP_MIN_LOG_LEVEL=3  # silence some TensorFlow warnings and logs.

import os
import cv2
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
from glob import glob
import nibabel as nib
import copy
```

```
env: TF_CPP_MIN_LOG_LEVEL=3  # silence some TensorFlow warnings and logs.
```

Set `IMAGE_DIR` to be the path of the **`3d_images`** folder.

[194]:
```
# FILL IN CODE HERE #

IMAGE_DIR = "/home/marchuo/assign1/data/lung/3d_images/"

# FILL IN CODE HERE #
```

## 2 Section 1: Data

There aren't that many fully annotated 3D images - let's see how many there are. Print out the paths of all the images (files beginning with "`IMG`") in the `IMGAGE_DIR`. You should see 4 images. We'll be using two of them for training, one for validation, and one for testing.

[195]:
```
# FILL IN CODE HERE #
# Hint - Use the glob function to list the files.
for name in glob(IMAGE_DIR + "IMG*"):
    print(name)
```

```
/home/marchuo/assign1/data/lung/3d_images/IMG_0031.nii.gz
/home/marchuo/assign1/data/lung/3d_images/IMG_0059.nii.gz
/home/marchuo/assign1/data/lung/3d_images/IMG_0002.nii.gz
/home/marchuo/assign1/data/lung/3d_images/IMG_0078.nii.gz
```

**Q1c.2**: Now let's load in the 3D images and their corresponding masks. Implement the `load_data` function according to the following guidelines - Load the image and its mask using `nib.load(path).get_fdata()`. There is an example here. - Downsample each image and mask by reading every 8 pixels, set in the `downscale` variable below. Make sure you only downsample along axis 1 and axis 2, and **not axis 0**. Each slice along axis 0 has a shape of `(512, 512)`. Working with high resolution slices would require more compututational resources, but we can get decent results by downsampling slices to `(64, 64)`. - Image pixel values represent Hounsfield units. Dense bones typically have values around ~2000HU and the values in our dataset range from about -2600 to +2600. The maximum value is 3071, so you need to normalize the image by dividing it by 3071. - Mask values are either 0 or 255. Scale this to the range [0,1] - Reshape the image and mask using `np.expand_dims` so that they both have shape `(N, 64, 64, 1)` - By default, the data are 64 bit floats. Use `np.astype` to convert the image to 32 bit float and the mask to 8 bit unsigned integer types.

[196]:
```
from scipy.ndimage import zoom
def load_data(uid):
    """
    Load a 3D image and its corresponding mask. Take a look at the next cell to␣
 ↪see
    how this function is called.

    Parameters:
    uid (int): A unique image and mask identifier. This is the number in the␣
 ↪file name.
                For instance, if the file name is `IMG_0002.nii.gz`, the `uid`␣
 ↪is 2.

    Returns:
    image (np.ndarray): A numpy array of shape (N, 64, 64, 1)
    mask (np.ndarray): A numpy array of shape (N, 64, 64, 1)
    """
    downscale = 8
    image_file = f"IMG_{uid:04}.nii.gz"
    mask_file = f"MASK_{uid:04}.nii.gz"

    # FILL IN CODE HERE #
    img = nib.load(IMAGE_DIR + image_file)
    mask = nib.load(IMAGE_DIR + mask_file)
    img_fdata = img.get_fdata()
    mask_fdata = mask.get_fdata()

    #Downsampling slices
    img_fdata = img_fdata[:, ::downscale, ::downscale]
    mask_fdata = mask_fdata[:, ::downscale, ::downscale]
    #img_fdata = zoom(img_fdata, (1, (1/8), (1/8)))

    #Normalize HU image
    img_fdata /= 3071

    #Scale Mask Data
    mask_fdata /= 255

    #Reshape image and mask
    img_fdata = np.expand_dims(img_fdata, axis=3)
    img_fdata = np.reshape(img_fdata,(img_fdata.shape[0], 64, 64, 1))
    mask_fdata = np.expand_dims(mask_fdata, axis=3)
    mask_fdata = np.reshape(mask_fdata,(mask_fdata.shape[0], 64, 64, 1))

    #change data type
    img_fdata = img_fdata.astype(np.float32)
    mask_fdata = mask_fdata.astype(np.uint8)

    image = img_fdata
    mask = mask_fdata
    # FILL IN CODE HERE #

    return image, mask
```

We will be using IMG_0002 and IMG_0031 as our training images, IMG_0059 as our validation image, and IMG_0078 as our test image. As a sanity check, make sure that the shapes match the expected output

[197]:
```
train_imgs, train_masks = zip(*[load_data(2), load_data(31)])
val_img, val_mask = load_data(59)
test_img, test_mask = load_data(78)

print(train_imgs[0].shape, train_masks[0].shape)  # (325, 64, 64, 1) (325, 64,␣
 ↪64, 1)
print(train_imgs[1].shape, train_masks[1].shape)  # (465, 64, 64, 1) (465, 64,␣
 ↪64, 1)
print(val_img.shape, val_mask.shape)              # (301, 64, 64, 1) (301, 64,␣
 ↪64, 1)
print(test_img.shape, test_mask.shape)            # (117, 64, 64, 1) (117, 64,␣
 ↪64, 1)
```

```
(325, 64, 64, 1) (325, 64, 64, 1)
(465, 64, 64, 1) (465, 64, 64, 1)
(301, 64, 64, 1) (301, 64, 64, 1)
(117, 64, 64, 1) (117, 64, 64, 1)
```

**Q1c.3**: Suppose we only had the budget to label approximately 30%~35% of the data, for example - only some 2D slices from the 3D volume will have ground truth masks. What strategy is used in the 3D U-Net paper to decide which 2D slices to label? Which dimensions do they sample along? Please describe this strategy for sampling 2D slices to label, and explain why it is the best method.

Hint - read section 3 of the paper.

*Written answer*: The annotation samples were chosen in accordance with good data representation - annotation slices were sampled as uniformly as possible across the 3 dimensions. They were sampled along the orthogonal xy, xz, and yz slices with 4 labels - “inside the tubule”, “tubule”, “background”, “unlabeled”, This is the best method to sample as there is representative labeling across all 3 dimensions, which better trains our model to generalize across all three dimensions on new test data. In this case, it is also best to sample along these axes, because we are interested in the three different lung views of its interior - axial, sagittal and coronal views.

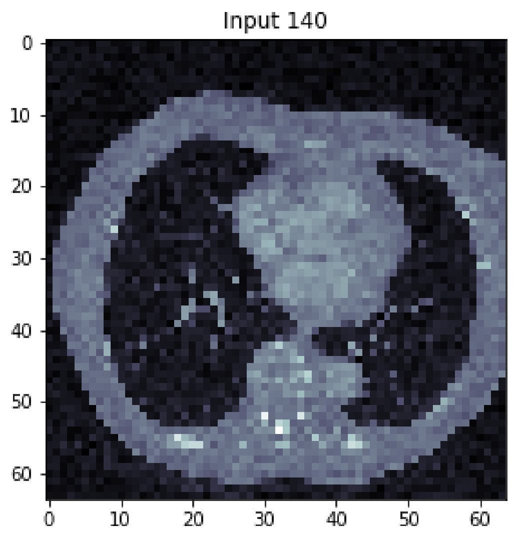
To simulate this semi-supervised scenario, we set all unlabelled voxels to be ignore_index, while the labelled 2D slices would be either 0 or 1 from the given mask. Implement the random sampling strategy from the paper in the `random_label` function. The following hints may be helpful: - sample_ratio is the ratio of slices you will sample from **each axis**. - unlabelled voxels should be set to `ignore_index`. - the starter code creates a `semi_supervised_mask` of the same size as the given mask with all the voxels set to `ignore_index`.

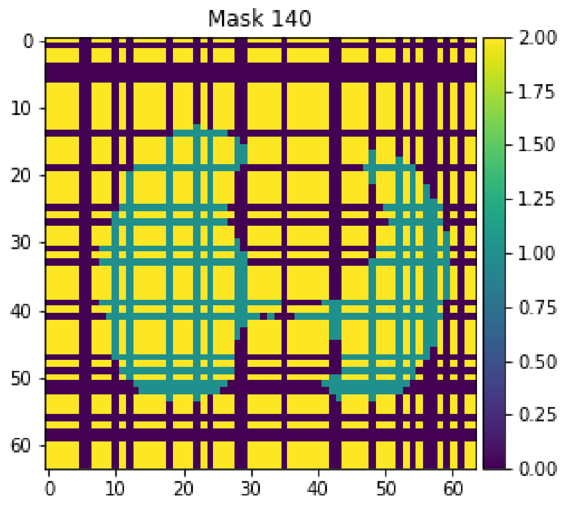
Example - Let's say you have a volume of size `(100, 100, 100)` with a sample ratio of 0.1. Then along each of the three axes, you'd sample roughly 10 planes and copy those labels from `mask` to `semi_supervised_mask`. The resulting `semi_supervised_mask` will have some labelled voxels (which will be 0 or 1) and some unlabelled voxels which are set to `ignore_index` (2 in this case).

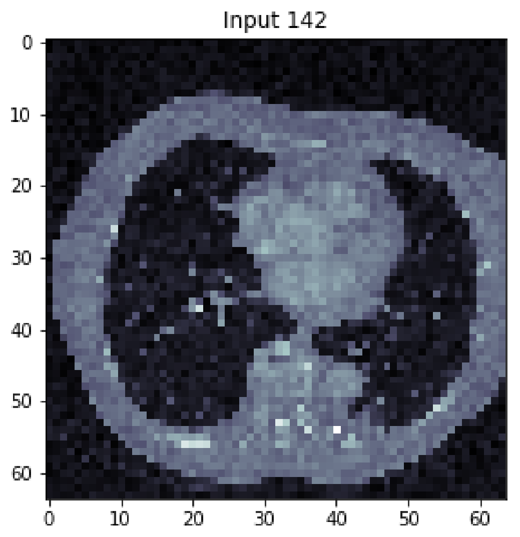
[198]:
```python
def random_fill(a, N, fillval=0):
    # a is input array
    # N is blocksize

    # Store shape info
    m,n,r = a.shape

    # Generate random start indices for second and third axes keeping proper
    # distance from the boundaries for the block to be accomodated within.
    idx0 = np.random.randint(0,n-N+1,m)
    idx1 = np.random.randint(0,r-N+1,m)

    # Iterate through first and use slicing to assign fillval.
    for i in range(m):
        a[i, idx0[i]:idx0[i]+N, idx1[i]:idx1[i]+N]  = fillval
    return a

def random_label(mask):
    """
    Simulates a semi-supervised scenario by retaining only a few labelled␣
 ↪slices from mask.

    Parameters:
    mask (np.ndarray): The mask array consisting of 0s and 1s

    Returns:
    semi_supervised_mask: The transformed mask array in which unablled pixels␣
 ↪are set to ignore_index.
    """
    ignore_index = 2
    sample_ratio = 0.3
    semi_supervised_mask = np.ones(mask.shape, mask.dtype) * ignore_index

    # FILL IN CODE HERE #
    #sample along xy, xz, and yz slices
    random_x_array = np.random.choice(mask.shape[0], round(mask.shape[0] *␣
↪sample_ratio), replace=False)
    random_y_array = np.random.choice(mask.shape[1], round(mask.shape[1] *␣
↪sample_ratio), replace=False)
    random_z_array = np.random.choice(mask.shape[2], round(mask.shape[2] *␣
↪sample_ratio), replace=False)

    for x in random_x_array:
        semi_supervised_mask[x, :, :] = mask[x, :, :]
    for y in random_y_array:
        semi_supervised_mask[:, y, :] = mask[:, y, :]
    for z in random_z_array:
        semi_supervised_mask[:, :, z] = mask[:, :, z]
    # FILL IN CODE HERE #

    return semi_supervised_mask

# Now we can apply this random sampling strategy to our train and validation␣
↪labels.
ss_train_masks = [random_label(mask) for mask in train_masks]
ss_val_mask = random_label(val_mask)
```

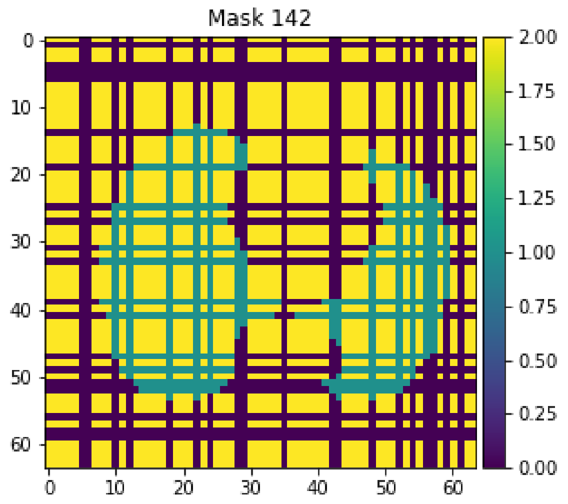
Let us visualize some slices of our data and mask arrays. Sparsely labelled slices have 3 values in the corresponding `mask` images - these represent the positive (1), negative (0), and unlabelled (2) voxels. Fully labelled slices have only two values in the `mask` images representing the positive and negative classes.

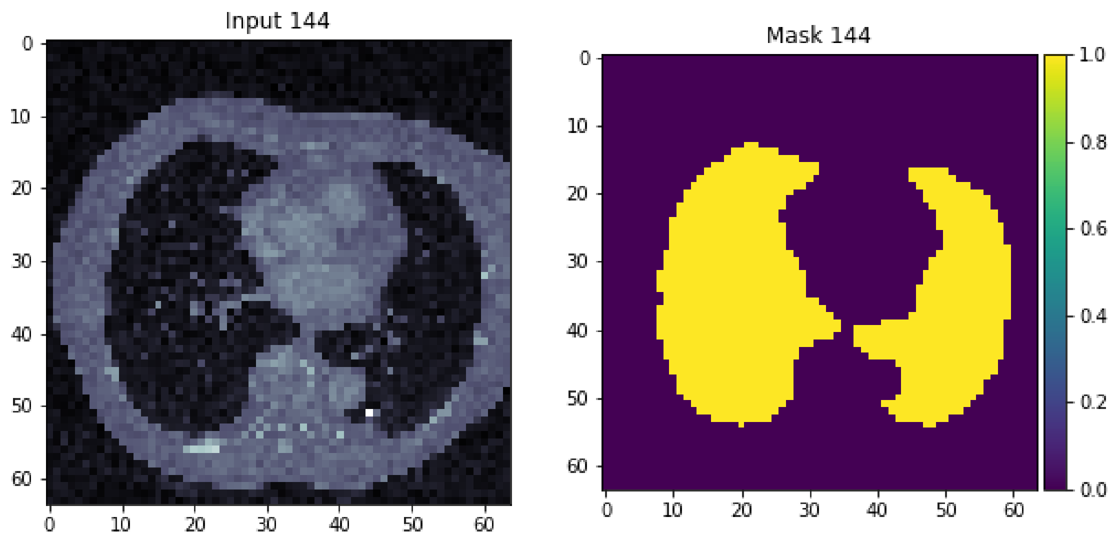
[199]:
```
for idx in range(140, 160, 2):
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (10, 5))
    ax1.imshow(val_img[idx], cmap = 'bone')

    im = ax2.imshow(ss_val_mask[idx])
    divider = make_axes_locatable(ax2)
    cax = divider.append_axes('right', size='5%', pad=0.05)
    fig.colorbar(im, cax=cax, orientation='vertical')

    ax1.set_title(f"Input {idx}")
    ax2.set_title(f"Mask {idx}")
```

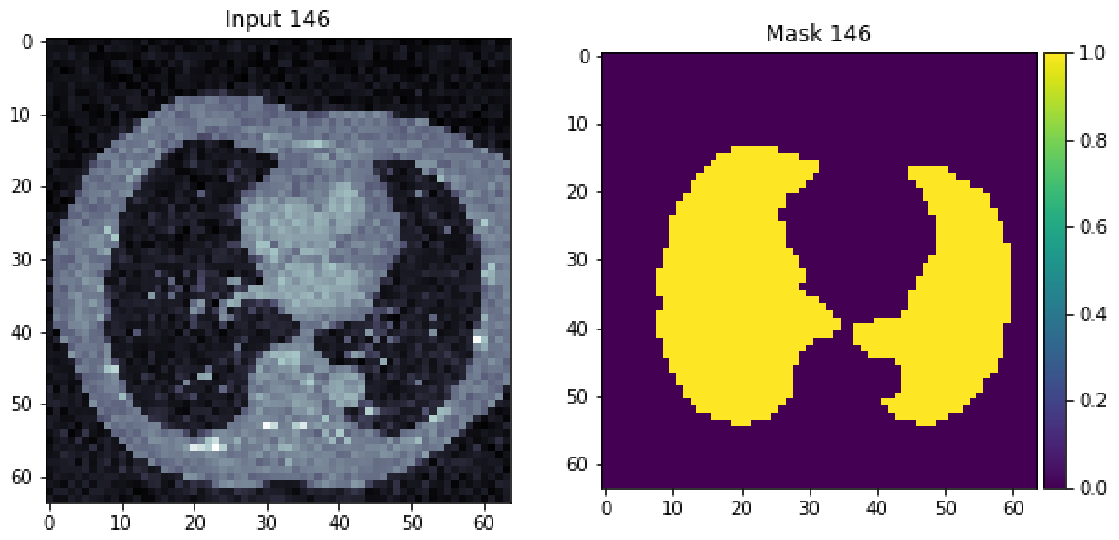

Input 140
0
10
20
30
40
50
60
0
10
20
30
40
50
60

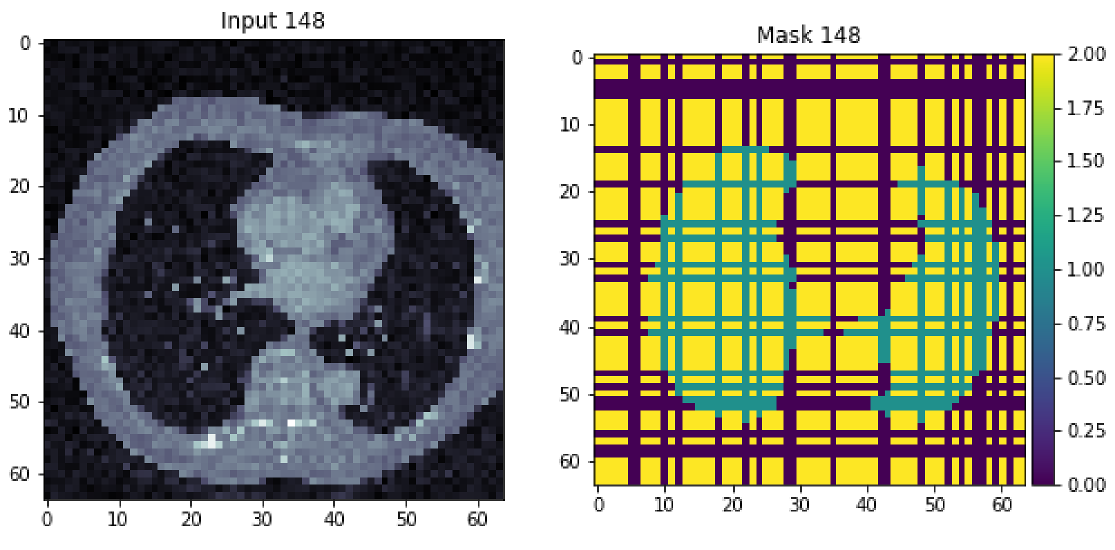

Mask 140
0
10
20
30
40
50
60
0
10
20
30
40
50
60
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00

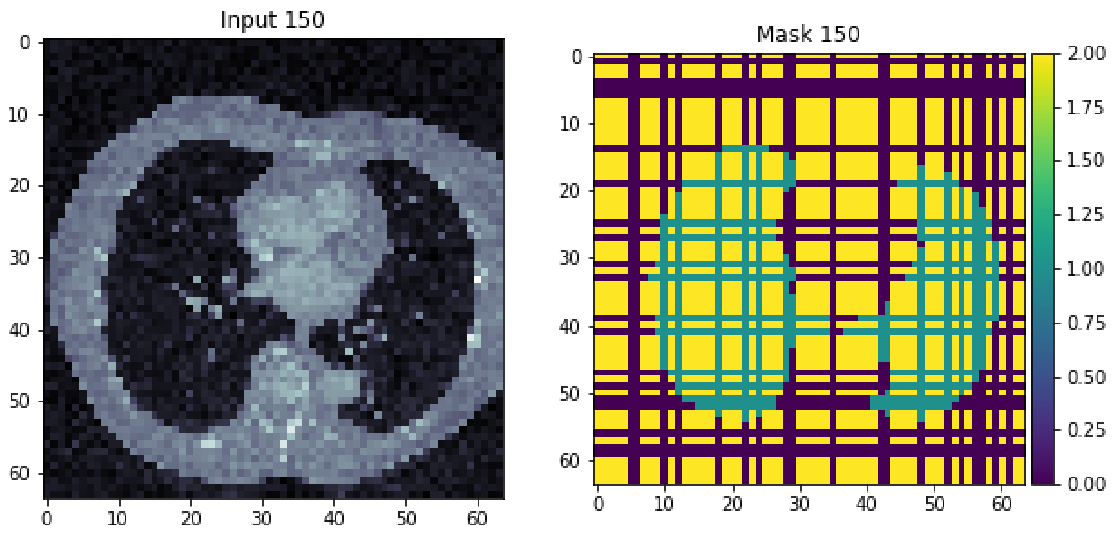

Input 142
0
10
20
30
40
50
60
0
10
20
30
40
50
60

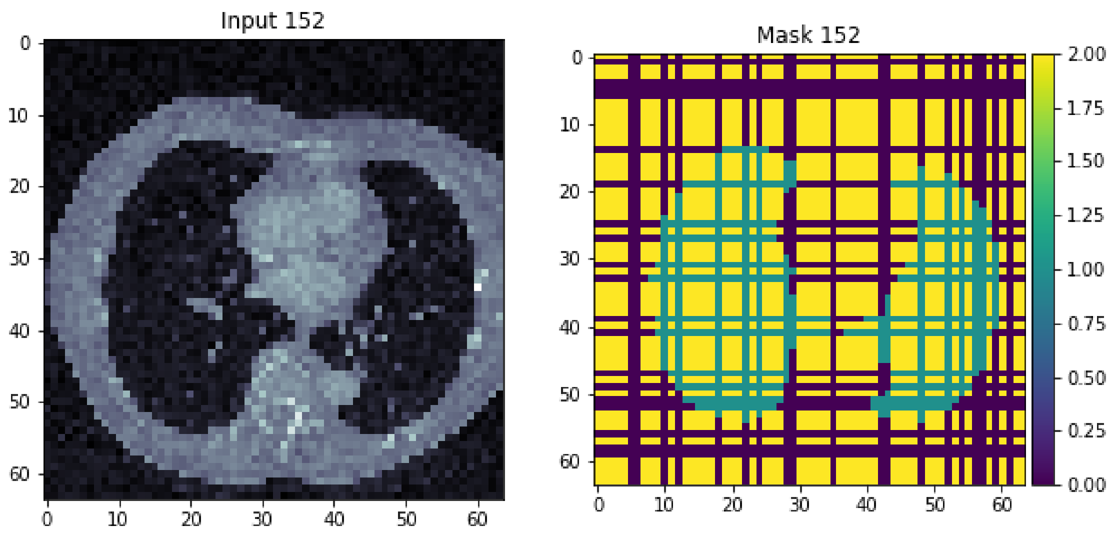

Mask 142
0
10
20
30
40
50
60
0
10
20
30
40
50
60
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00

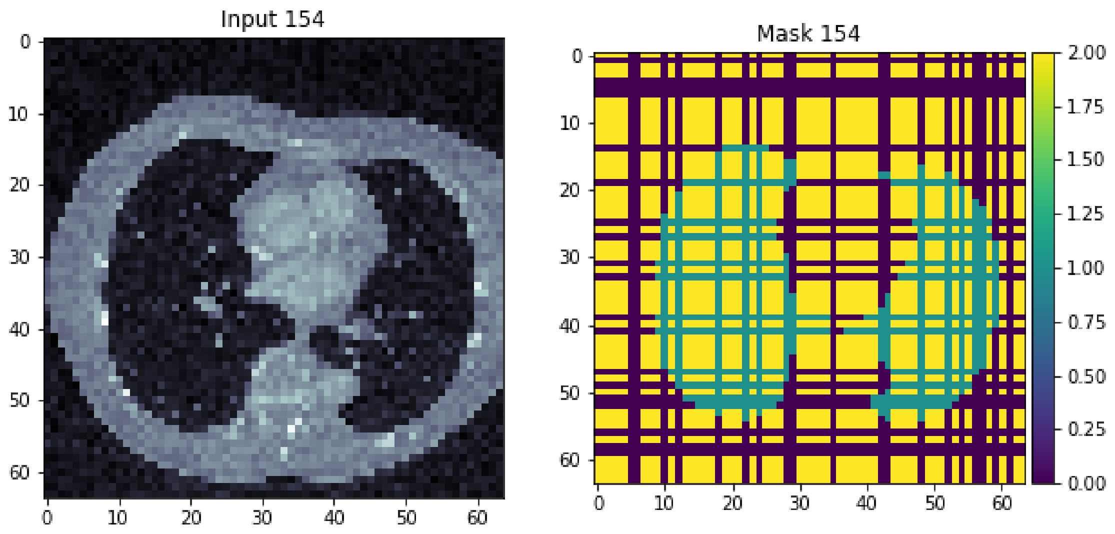

Input 144
Mask 144
0
10
20
30
40
50
60
0.0
0.2
0.4
0.6
0.8
1.0

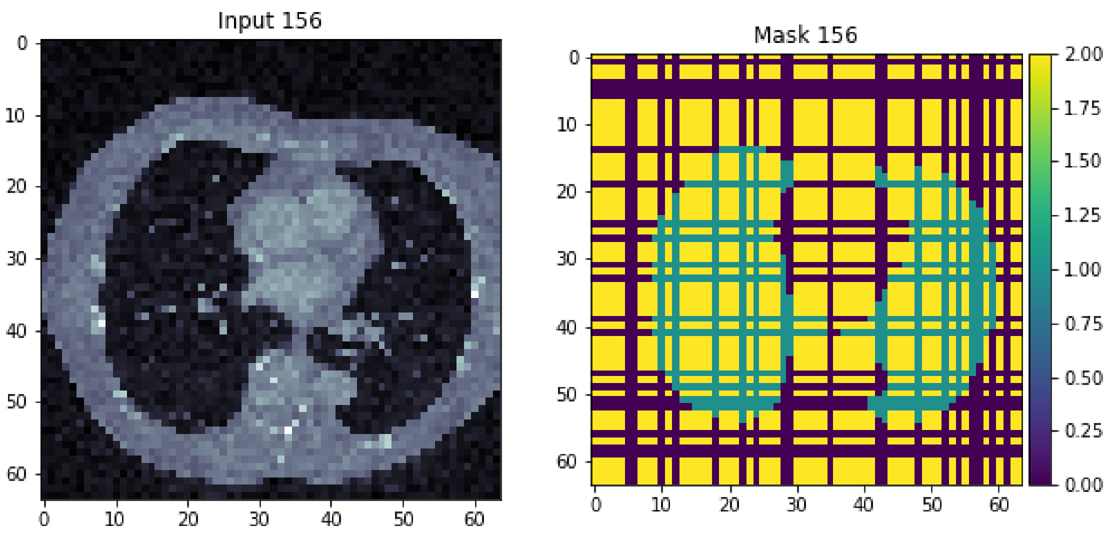

Input 146
Mask 146
0
10
20
30
40
50
60
0.0
0.2
0.4
0.6
0.8
1.0

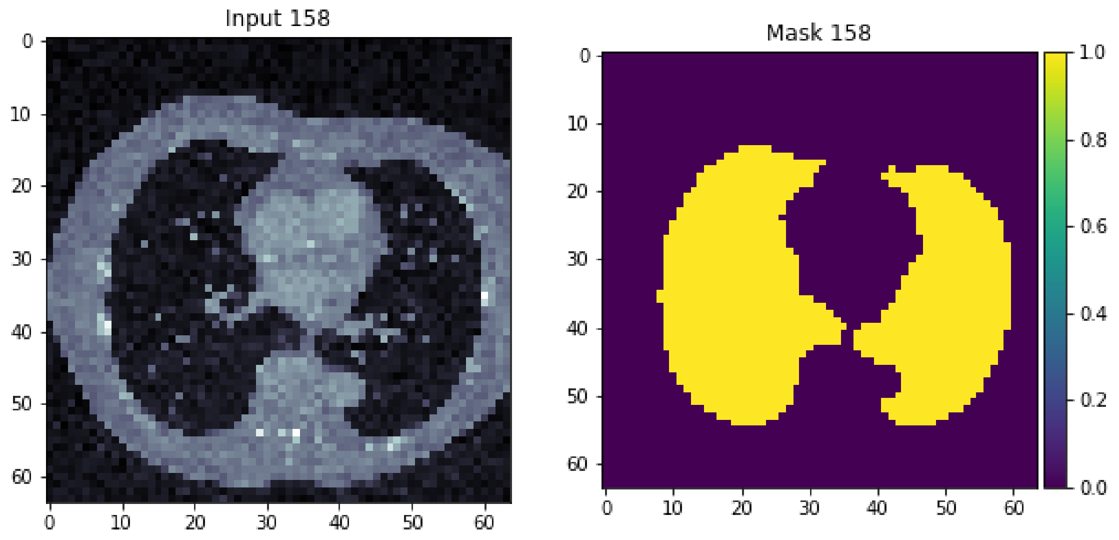

Input 148
Mask 148
0
10
20
30
40
50
60
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00


Input 150
Mask 150
0
10
20
30
40
50
60
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00


Input 152
Mask 152
0
10
20
30
40
50
60
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00


Input 154
Mask 154
0
10
20
30
40
50
60
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00





**Q1c.4**: While working with 3D biomedical data, we often encounter high resolution voxel data which may not fit in our GPU memory. One solution is to break a large volume into smaller chunks and use them for training. To implement this solution, we will build a generator function which samples smaller volumes from our data.

The `generate_chunk` function yields sub volumes of size (64, 64, 64, 1) from a list of input 3D images. For instance, if we provide a list with our two training images, `generate_chunk` will first pick one of them at random (weighted by the size of each image, which may be different), and will then sample a sub-volume from that image. The first part, picking an image from the list, has already been implemented for you. You need to implement the chunk sampling code.

Your chunk sampling code needs to sample an image chunk and its corresponding semi-supervised mask along **axis 0**. Using the semi-supervised mask, you will compute a `sample_weight` array which indicates if a voxel is labelled or not. This will be used later by the model to weight the loss function so that unlabelled voxels are ignored while computing the loss. Note that the `sample_weight` array will have **1 fewer dimension** than the image and mask. So its shape will be `(64, 64, 64)`.

Fill in the `generate_chunk` function to yield an `image_chunk`, its corresponding `mask_chunk`, and the resulting `sample_weight`. Voxels which are labelled should have a `sample_weight` value of 1 and unlabelled voxels should be assigned a value of 0.

[206]:
```
import itertools
def build_generator_fn(images, masks, slice_count=64):
    """
    Returns a data generator function which can be fed to a tf.data.Dataset␣
↪object.

    Parameters:
    images (List[np.ndarray]): A list of 3D images of shape (N, 64, 64, 1). `N`␣
↪may be different for each image.
    masks (List[np.ndarray]): A list of corresponding masks of shape (N, 64,␣
↪64, 1). `N` may be different for each image.
    slice_count (int): The number of slices to sample along axis 0 of each␣
↪image.

    Returns:
    A generator function with no parameters.

    """
    def generate_chunk():
        """
        Yields a random image chunk, corresponding mask chunk, and the computed␣
↪sample weight.
        The sample_weight for a voxel is 1 when the mask value is 0 or 1.
        The sample_weight for a voxel is 0 when the mask value is␣
↪`ignore_index` (2 in this instance)

        Yields:
        image_chunk (np.ndarray): A numpy array of shape (64, 64, 64, 1)
        mask_chunk (np.ndarray): A numpy array of shape (64, 64, 64, 1)
        sample_weight (np.ndarray): A numpy array of shape (64, 64, 64)
        """
        while True:
            # First we pick an image at random from the list.
            num_slices = np.array([img.shape[0] for img in images])
            idx = np.random.choice(len(images), p=num_slices / num_slices.
↪sum())  # weighted random choice
            image, mask = images[idx], masks[idx]
            # FILL IN CODE HERE #
            xsize, ysize, zsize, label = image.shape
            xbatch, ybatch, zbatch = (slice_count, slice_count, slice_count)
            i = np.random.randint(0, xsize-xbatch)
            image_chunk = image[i: i + xbatch,
                                :,
                                :]
            mask_chunk = mask[i: i + xbatch,
                               :,
                               :]
            arr = np.copy(mask_chunk[:, :, :, :])
            where_0 = np.where(arr == 0)
            where_1 = np.where(arr == 1)
            where_2 = np.where(arr == 2)
            arr[where_0] = 1
            arr[where_1] = 1
            arr[where_2] = 0
            sample_weight = arr[:, :, :, 0]

            # FILL IN CODE HERE #
            yield image_chunk, mask_chunk, sample_weight
    return generate_chunk

train_generator = build_generator_fn(train_imgs, ss_train_masks)
val_generator = build_generator_fn([val_img], [ss_val_mask])

img, mask, sample_w = next(train_generator())
print(img.shape, mask.shape, sample_w.shape)  # (64, 64, 64, 1) (64, 64, 64, 1)␣
 ↪(64, 64, 64)
assert sample_w.shape == (64, 64, 64), "Shape mismatch, did you remove the last␣
 ↪channel?"
```

```
(64, 64, 64, 1) (64, 64, 64, 1) (64, 64, 64)
```

**Q1c.5**: Now that we have our data generators set up, we will use them to create Dataset objects. We will use the `from_generator` function to create our train and validation datasets. You need to:

- Define the `output_signature` variable. In this instance, it is a tuple of 3 tf.TensorSpec objects, one for each of the 3 values yielded by the generator. Each TensorSpec object defines a shape and a datatype. For example, the TensorSpec for the `sample_weight` is `tf.TensorSpec(shape=(None, 64, 64), dtype=tf.float32)`. Note that the shape along axis 0 is None, this needs to be done for the other two TensorSpecs as well. This will let the model handle inputs of arbitrary size along axis 0.
- Create a Dataset object from the `generator` by providing the `output_signature`.
- Batch the inputs according to `batch_size` and prefetch two batches. Click the links for documentation.

```
[207]: def create_dataset(generator, batch_size=8):
           """
           Creates a TensorFlow Dataset using the provided generator

           Parameters:
           generator (generator): A callable function which yields a tuple of inputs,␣
        ↪labels, and sample weight.
           batch_size (int): The number of generated inputs to be batched together.

           Returns:
           dataset (tf.data.Dataset): A dataset object wrapped around the generator,␣
        ↪which batches inputs according
                                           to `batch_size` and prefetches 2 batches.
           """
           # FILL IN CODE HERE #
           dataset = tf.data.Dataset.from_generator(generator, output_signature=(
               tf.TensorSpec(shape=(None, 64, 64, 1), dtype=tf.float32),
               tf.TensorSpec(shape=(None, 64, 64, 1), dtype=tf.uint8),
               tf.TensorSpec(shape=(None, 64, 64), dtype=tf.float32)))
           dataset = dataset.batch(batch_size).prefetch(2)
           # FILL IN CODE HERE #

           return dataset

       train_dataset = create_dataset(train_generator)
       val_dataset = create_dataset(val_generator)
       data = list(train_dataset.take(1))
       data[0][0].shape  # TensorShape([8, 64, 64, 64, 1])
```

```
[207]: TensorShape([8, 64, 64, 64, 1])
```

## 3 Section 2: Model

**Q1c.6**: We will be using the 3D U-Net model to train and run inference in 3D, with the paper linked here https://arxiv.org/abs/1606.06650. 3D U-Net is a cool extension to the 2D U-Net that enables the model to work with 3D data. Please read the paper, describe the two use cases briefly, and point out which one we are using in this case.

*Written answer*: 1) Semi-automated setup: The user annotates some slices in the volume to be segmented, and the network trains from those sparse annotations to provide a dense 3D segmentation. It is assumed that the train dataset has not previously been annotated - the user needs domain expertise in being able to annotate the slices.

2) Fully-automated setup: A representative sparsely annotated training dataset exists - the network trains on this data and the network densely segments new volumetric data. It is assumed that the user wants to segment a large number of images in a comparable setting. The paper trains on two partially annotated kidney volumes and uses the trained model to segment on the third.

We will be using the 2nd use case, since we have a representative sparsely annotated training dataset that we artificially created from a ground truth mask.

**Q1c.7**: Now let's define a 3D version of the U-Net model. A good place to start would be to copy your 2D U-Net architecture from part b and change every operation to its 3D counterpart to create the 3D U-Net! Note that the 3D U-Net paper makes some additional minor changes that we won't implement here.

Please stay within the following constraints: - Use only these layers: `Conv3D`, `MaxPool3D` and `Upsampling3D`. You will also use `concatentate` - The architecture (the order of `Conv3D`, `MaxPool3D`, and `Upsampling3D` should be similar to the 3D U-Net architecture.

Here are some guidelines for an architecture that worked well: - **Downsampling / Upsampling steps**: 2 `MaxPool3D` layers on the left branch, and 2 `Upsampling3D` layers on the right branch. The original paper has 3 each (red and yellow arrows in Fig 2). - **Conv Layers**: Our model uses just one `Conv3D` layer per level, for a total of 3 on the left branch and 2 on the right branch. The original paper stacks 2 `Conv3D` layers (orange arrows in Fig 2) at each level. We suggest filter counts of [64, 128, 256] on the left branch and [128, 64] on the right branch. - **UpConv Layers**: In the original paper, the `Conv3D` layer applied immediately after `UpSampling3D` preserves the number of channels. Our model halves the number of channels so that equal sized inputs are concatenated (`128 + 128` and `64 + 64` instead of `128 + 256` and `64 + 128` in Fig 2) - **Final Layer** - Same as the one mentioned in the paper. - **Batch Norm** - We do not use any `BatchNormalization` layers.

[208]:
```
from tensorflow.keras.layers import Conv3D, MaxPool3D, UpSampling3D, concatenate

class U_Net(tf.keras.Model):

    def __init__(self):
        super(U_Net, self).__init__()

        # FILL IN CODE HERE #
        self.conv1 = Conv3D(64, (3, 3, 3), padding="same", activation='relu')␣
↪#concat

        self.pool1 = MaxPool3D(pool_size=(2, 2, 2))

        # Second conv layer
        self.conv2 = Conv3D(128, (3, 3, 3), padding="same", activation='relu')␣
↪#concat

        self.pool2 = MaxPool3D(pool_size=(2, 2, 2))

        # Third conv layer
        self.conv3 = Conv3D(256, (3, 3, 3), padding="same", activation='relu')

        self.upsample1 = UpSampling3D((2, 2, 2))


        self.upconv1 = Conv3D(128, (3, 3, 3), padding="same", activation='relu')
        self.upsample2 = UpSampling3D((2, 2, 2))

        self.upconv2 = Conv3D(64, (2, 2, 2), padding="same", activation='relu')

        self.output_layer = Conv3D(1, (1, 1, 1), activation='sigmoid')
        # FILL IN CODE HERE #

    def call(self, inputs):
        # FILL IN CODE HERE #
        conv1 = self.conv1(inputs)

        output = self.pool1(conv1)

        conv2 = self.conv2(output)

        output = self.pool2(conv2)

        output = self.conv3(output)

        upsample1 = self.upsample1(output)

        output = concatenate([upsample1, conv2], axis=-1)

        output = self.upconv1(output)

        upsample2 = self.upsample2(output)

        output = concatenate([upsample2, conv1], axis=-1)

        output = self.output_layer(output)
        # FILL IN CODE HERE #

        return output

model = U_Net()
```

**Q1c.8**: Compile and train the model for 20 epochs with `model.fit`. Set `steps_per_epoch` to be 20, `validation_steps` to be 20. In our experiments, a model implementing the suggested architecture achieves a validation loss of just under 0.04. It takes about 3 minutes per epoch on a K80 GPU.

Recall that in the semi-supervised setting, we do not compute the loss on unlabelled slices. This is achieved by providing a `sample_weight` tensor to the loss function in addition to the predicted and true values. Our `Dataset` object generates these sample weights for us (refer to **Q1c.4**) so we don't need to set the `sample_weight` parameter in `model.fit()`. See `https://www.tensorflow.org/guide/keras/train_and_evaluate#sample_weights` for a simplified example.

[209]:
```
STEPS_PER_EPOCH = 20
VALIDATION_STEPS = 20
EPOCHS = 20

# FILL IN CODE HERE #
model.compile('adam', 'binary_crossentropy', 'accuracy')
#model.summary()
model.fit(train_dataset, epochs = EPOCHS, validation_data=val_dataset,␣
 ↪validation_steps = VALIDATION_STEPS,
          steps_per_epoch=STEPS_PER_EPOCH)
# FILL IN CODE HERE #
```

```
Epoch 1/20
20/20 [==============================] - 73s 4s/step - loss: 0.3182 - accuracy:
0.5453 - val_loss: 0.3217 - val_accuracy: 0.5231
Epoch 2/20
20/20 [==============================] - 71s 4s/step - loss: 0.1795 - accuracy:
0.5579 - val_loss: 0.1630 - val_accuracy: 0.5281
Epoch 3/20
20/20 [==============================] - 71s 4s/step - loss: 0.0978 - accuracy:
0.5842 - val_loss: 0.1121 - val_accuracy: 0.6235
Epoch 4/20
20/20 [==============================] - 71s 4s/step - loss: 0.0801 - accuracy:
0.6269 - val_loss: 0.0765 - val_accuracy: 0.6272
Epoch 5/20
20/20 [==============================] - 71s 4s/step - loss: 0.0473 - accuracy:
0.6355 - val_loss: 0.0640 - val_accuracy: 0.6271
Epoch 6/20
20/20 [==============================] - 71s 4s/step - loss: 0.0336 - accuracy:
0.6395 - val_loss: 0.0494 - val_accuracy: 0.6327
Epoch 7/20
20/20 [==============================] - 71s 4s/step - loss: 0.0312 - accuracy:
0.6400 - val_loss: 0.0451 - val_accuracy: 0.6348
Epoch 8/20
20/20 [==============================] - 71s 4s/step - loss: 0.0292 - accuracy:
0.6377 - val_loss: 0.0434 - val_accuracy: 0.6353
Epoch 9/20
20/20 [==============================] - 71s 4s/step - loss: 0.0290 - accuracy:
0.6374 - val_loss: 0.0432 - val_accuracy: 0.6335
Epoch 10/20
20/20 [==============================] - 71s 4s/step - loss: 0.0285 - accuracy:
0.6401 - val_loss: 0.0418 - val_accuracy: 0.6364
Epoch 11/20
20/20 [==============================] - 71s 4s/step - loss: 0.0277 - accuracy:
0.6369 - val_loss: 0.0408 - val_accuracy: 0.6364
Epoch 12/20
20/20 [==============================] - 71s 4s/step - loss: 0.0259 - accuracy:
```

```
0.6415 - val_loss: 0.0408 - val_accuracy: 0.6379
Epoch 13/20
20/20 [==============================] - 71s 4s/step - loss: 0.0260 - accuracy:
0.6423 - val_loss: 0.0416 - val_accuracy: 0.6372
Epoch 14/20
20/20 [==============================] - 71s 4s/step - loss: 0.0254 - accuracy:
0.6407 - val_loss: 0.0387 - val_accuracy: 0.6352
Epoch 15/20
20/20 [==============================] - 71s 4s/step - loss: 0.0261 - accuracy:
0.6399 - val_loss: 0.0392 - val_accuracy: 0.6350
Epoch 16/20
20/20 [==============================] - 71s 4s/step - loss: 0.0246 - accuracy:
0.6374 - val_loss: 0.0438 - val_accuracy: 0.6378
Epoch 17/20
20/20 [==============================] - 71s 4s/step - loss: 0.0249 - accuracy:
0.6411 - val_loss: 0.0406 - val_accuracy: 0.6350
Epoch 18/20
20/20 [==============================] - 71s 4s/step - loss: 0.0236 - accuracy:
0.6363 - val_loss: 0.0378 - val_accuracy: 0.6370
Epoch 19/20
20/20 [==============================] - 71s 4s/step - loss: 0.0237 - accuracy:
0.6437 - val_loss: 0.0394 - val_accuracy: 0.6376
Epoch 20/20
20/20 [==============================] - 71s 4s/step - loss: 0.0236 - accuracy:
0.6418 - val_loss: 0.0408 - val_accuracy: 0.6349
```

[209]:
```
<keras.callbacks.History at 0x7f2d14271c90>
```

## 4 Section 3: Evaluation

Let's evaluate our 3D U-Net using the same metric as our 2D U-Net! Copy over the `compute_IoU` function from part b.

[210]:
```python
def compute_IoU(target, prediction):
    """
    Evaluate the intersection over union score for a given prediction and␣
 ↪ground truth value

    Parameters:
    target: (np.ndarray) : The ground truth label values
    prediction (np.ndarray): The labels predicted by the model

    Returns:
    iou (float) the IOU score
    """
    # FILL IN CODE HERE #
    intersection = np.logical_and(target, prediction)
    union = np.logical_or(target, prediction)
    iou = np.sum(intersection) / np.sum(union)
    # FILL IN CODE HERE #

    return iou
```

**Q1c.9**: Before you can perform inference on the test image, you need to reshape and pad it. The test image has dimensions (117, 64, 64, 1). The output shape of the 3D U-Net will not be the same as the input shape if any of the dimensions (except the channel dimension) are not divisible by `2^N` where `N` is the number of downsampling / upsampling layers.

The model makes predictions on a batch of inputs. So the padded image needs to be reshaped to represent a batch with a single input. Implement the padding and reshaping operations in the `pad_and_reshape_image` function. We will use this function to pad and reshape the test image. Note that we will only be padding along **axis 0** of the image. The padded regions should be filled with **zeros**.

```
[231]: def pad_and_reshape_image(image, new_dim):
           """
           Zero pads the image along axis 0, and then reshapes it into a batch of 1␣
       ↪image.

           Parameters:
           image (np.ndarray) - the input 3D image of shape (N, 64, 64, 1)
           new_dim (int) - the new shape along axis 0. new_dim > N

           Returns:
           padded_image (np.ndarray) - the padded and reshaped image of shape (1,␣
       ↪new_dim, 64, 64, 1)
           """

           # FILL IN CODE HERE #
           padding = new_dim - image.shape[0]
           npad = [(0, 0)] * image.ndim
           npad[0] = (0, padding)
           image = np.pad(image, pad_width=npad, mode='constant', constant_values=0)
           padded_image = image[np.newaxis, :, :, :, :]
           # FILL IN CODE HERE #

           return padded_image

       padded_test_image = pad_and_reshape_image(test_img, 128)
       print(padded_test_image.shape)  # (1, 128, 64, 64, 1)
```

```
(1, 128, 64, 64, 1)
```

**Q1c.10**: Now let's calculate the IOU on the test image similarly to part b. Report your score below; you should get a mean IOU of at least 0.75 (our model has a score of 0.9).

- Your model should make predictions on the `padded_test_image`.
- You should not compute the IOU over the padded regions.
- You need to slice and reshape your model's predictions to be the same shape as `test_mask`.
- Calculate the IOU with the threshold set as 0.5.

[242]:
```
# FILL IN CODE HERE #
pred = model.predict(padded_test_image)
thresh = 0.5

pred_copy = np.copy(pred)
bin_img = np.where(pred_copy > thresh, 1, 0)
bin_img = bin_img[0, :test_mask.shape[0], :, :, :]
iou = compute_IoU(test_mask, bin_img)
print(iou)
# FILL IN CODE HERE #
```

```
0.8308130699088145
```

**Q1c.11**: Finally, let's plot out an example of the test set, with input, prediction, and ground truth side by side. You should reshape your model's prediction to have shape `(117, 64, 64)`. Set `test_pred` to be the **reshaped** prediction from your model.

[243]:
```
test_img_sq = test_img.squeeze()      # (117, 64, 64)
test_mask_sq = test_mask.squeeze()    # (117, 64, 64)
test_pred = bin_img[:, :, :, 0]       # (117, 64, 64)

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize = (20, 7))
ax1.imshow(np.flip(test_img_sq.mean(1), 0), cmap = 'bone')
ax1.set_aspect(0.5)
ax2.imshow(np.flip(test_pred.sum(1), 0), cmap = 'bone')
ax2.set_title('Prediction')
ax2.set_aspect(0.5)
ax3.imshow(np.flip(test_mask_sq.sum(1), 0), cmap = 'bone')
ax3.set_title('Ground truth')
ax3.set_aspect(0.5)
```