A1_part3_3d_lung_segmentation

October 20, 2022

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 3D structure with 3D convolutions. This is useful since 3D voxel 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 that has a much smaller number of complete 3D scans instead of 2D scans. Our goal is to conduct volumetric 3D segmentation on our test set.

Labelling a complete 3D scan is time intensive and we will rarely have complete 3D scans at hand. In this assignment, we'll see whether training with a sparse training set can still allow us to approach the accuracy of training a complete, dense train set. In our case, sparsity refers to having labels for only a small portion of the voxels in our 3D volume, for example, only haveing labels for 30% of 2D slices, while the other 70% is unlabelled. We'll be simulating this scenario below, which is an example 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.

Semi supervised learning is learning from datasets that are partially labeled (small amount of labeled data + larger amount of unlabelled data).

Two Examples: 1) Automatic brain tumor segmentation. Inputs - parially labeled brain images; some brains are analyzed by clinicians and the tumors are annotated and other brain images have no annotations. Outputs - a prediction mask where pixels are labeled as 0 or 1 (non tumor or tumor). 2) Gene expression profiling. Inputs - partially labaled genes; some genes are manually labeled by experts and others are unlabeled. Outputs - a psuedo label for each unlabeled gene.

[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 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.

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


IMAGE_DIR = "/home/jupyter/assign1/3d_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 `IMAGE_DIR`. You should see 4 images. We'll be using two of them for training, one for validation, and one for testing.

[3]:
```
# FILL IN CODE HERE #
# Hint - Use the glob function to list the files.
file_list = glob(IMAGE_DIR + 'IMG*')
print('file_list {}'.format(file_list))
```

```
file_list ['/home/jupyter/assign1/3d_images/IMG_0031.nii.gz',
'/home/jupyter/assign1/3d_images/IMG_0078.nii.gz',
'/home/jupyter/assign1/3d_images/IMG_0002.nii.gz',
'/home/jupyter/assign1/3d_images/IMG_0059.nii.gz']
```

Q1c.2: Now let's load 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 imagedtype='float32' 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.

[4]:
```
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 #
    image = nib.load(IMAGE_DIR + image_file).get_fdata()/3071
    mask = nib.load(IMAGE_DIR + mask_file).get_fdata()/255
    image = image[:,0::8, 0::8]
    mask = mask[:,0::8, 0::8]
    image = np.expand_dims(image, axis= 3)
    mask = np.expand_dims(mask, axis = 3)
    image = image.astype(dtype='float32')
    mask = mask.astype(dtype='uint8')




    # 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

[5]:
```
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.

In the paper, they manually annotated some orthogonal xy, xz, and yz slices in each volume using a tool, Slicer3D2. The annotation positions were selected according to good data representation (i.e. annotation slices were sampled as uniformly as possible in all three dimensions). They sampled along all dimensions (x, y, z). They ran all their experiments on down-sampled version of the original image by a factor of two in each dimension.

This method is optimal for multiple reasons. First, it is labor-intensive to provide ground truth 3D annotation, so sparse 2D annotations (xy, xz, and yz annotated slices) are preferred. Additionally, since neighboring slices often share similar information, using sparse annotations leverages the fact that each sample contains many instances of same repetitive structures.

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`.

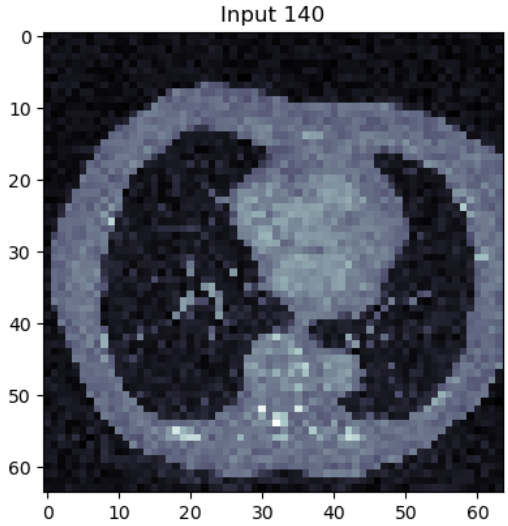
Example - Let's say you have a volume of size `(100, 100, 100)` with a sample ratio of 0.1. Then along each of the three axes, you'd sample roughly 10 planes and copy those labels from `mask` to `semi_supervised_mask`. The resulting `semi_supervised_mask` will have some labelled voxels (which will be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

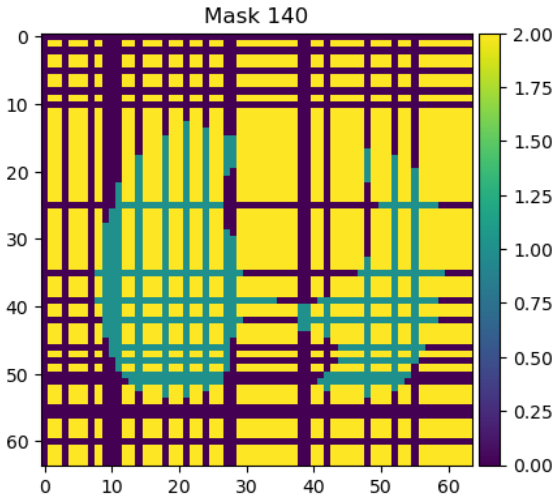
[6]:
```
def random_label(mask):
    """
    Simulates a semi-supervised scenario by retaining only a few labelled␣
↪slices from mask.

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

    Returns:
    semi_supervised_mask: The transformed mask array in which unlablled pixels␣
↪are set to ignore_index.
    """
```

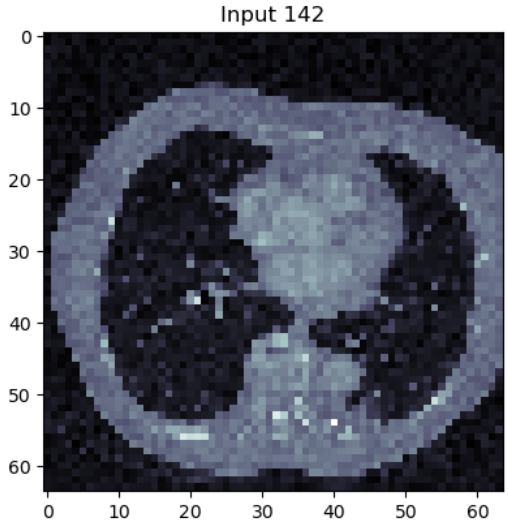
```
    ignore_index = 2
    sample_ratio = 0.3
    semi_supervised_mask = np.ones(mask.shape, mask.dtype) * ignore_index

    # FILL IN CODE HERE #
    dim1, dim2, dim3, not_needed = np.shape(semi_supervised_mask)
    samp1 =  np.random.randint(dim1, size=int(sample_ratio * dim1))
    samp2 =  np.random.randint(dim2, size=int(sample_ratio * dim2))
    samp3 =  np.random.randint(dim3, size=int(sample_ratio * dim3))

    semi_supervised_mask[samp1, :, :] = mask[samp1, :, :]
    semi_supervised_mask[:, samp2, :] = mask[:, samp2, :]
    semi_supervised_mask[:, :, samp3] = mask[:, :, samp3]
    # 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)
```

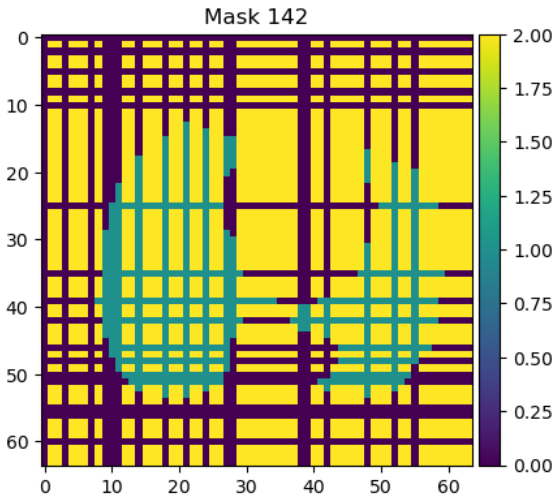
Let's visualize some slices of our image and mask arrays. Sparsely labelled slices have 3 values in their corresponding `mask` images - these represent the positive (1), negative (0), and unlabelled (2) voxels. Fully labelled slices have only two values in their `mask` images, representing the positive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

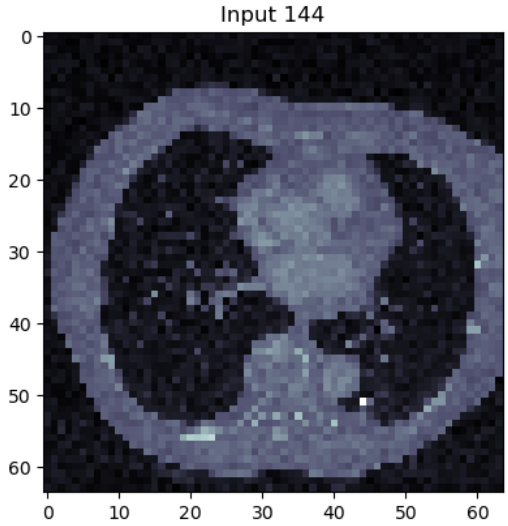
[7]:
```
for idx in range(140, 160, 2):
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (10, 5))
    ax1.imshow(val_img[idx], cmap = 'bone')

    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}")
```

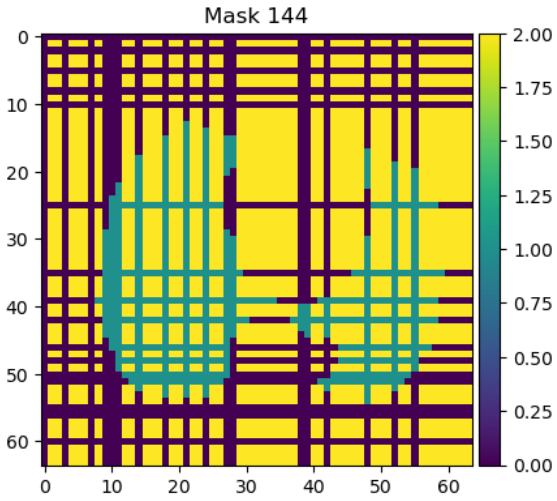

Input 140
0
10
20
30
40
50
60

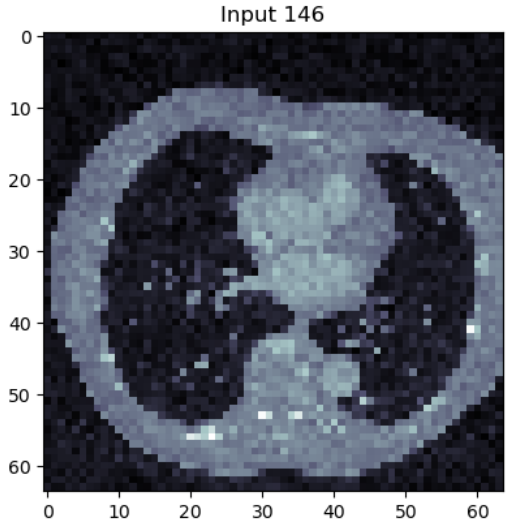

Mask 140
0
10
20
30
40
50
60
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00

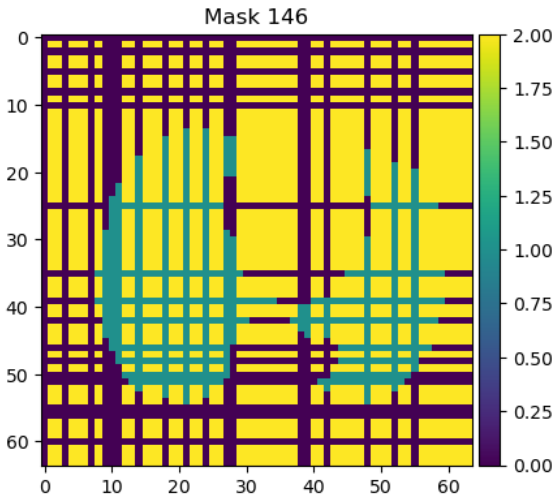

Input 142
0
10
20
30
40
50
60

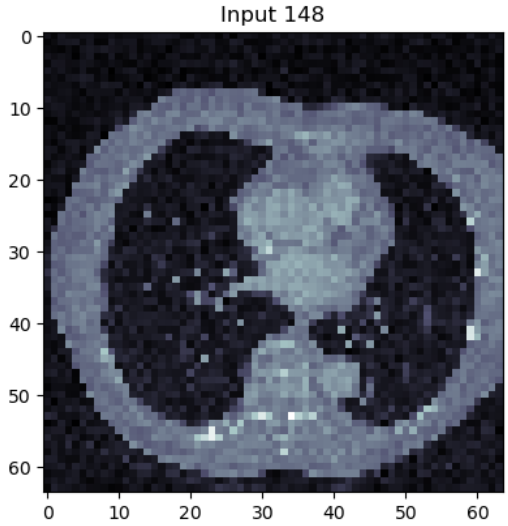

Mask 142
0
10
20
30
40
50
60
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00

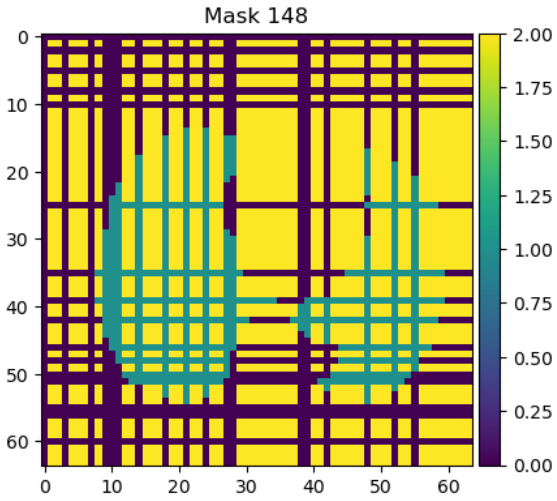

Input 144
0
10
20
30
40
50
60

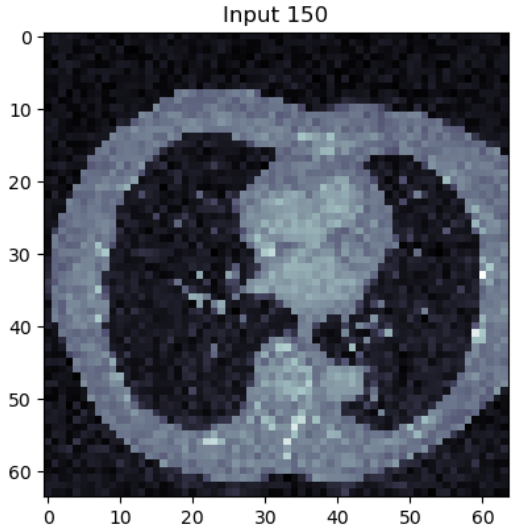

Mask 144
0
10
20
30
40
50
60
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00

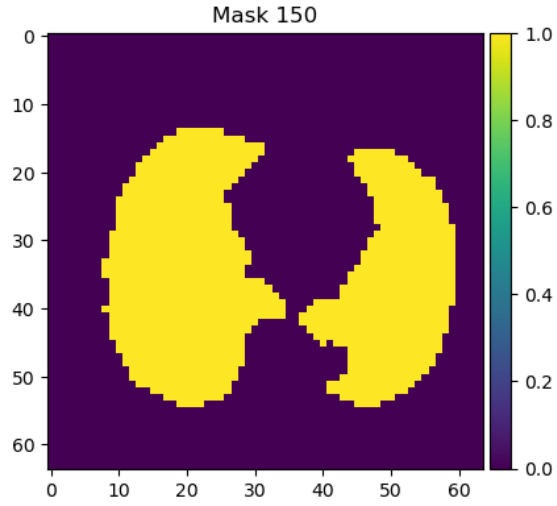

Input 146
0
10
20
30
40
50
60

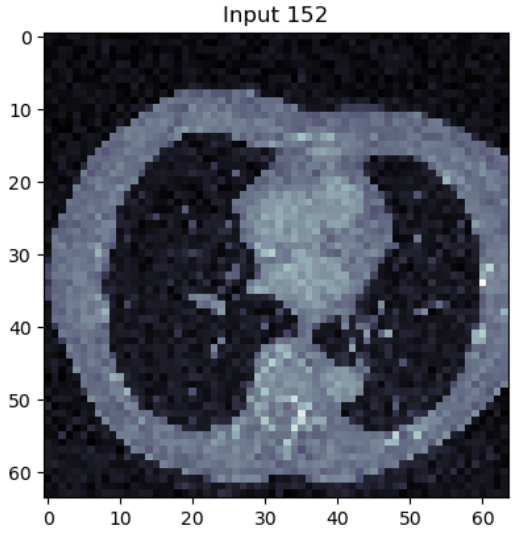

Mask 146
0
10
20
30
40
50
60
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00

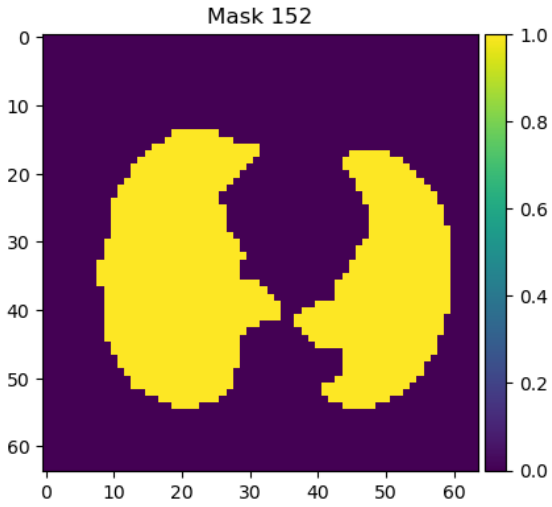

Input 148
0
10
20
30
40
50
60

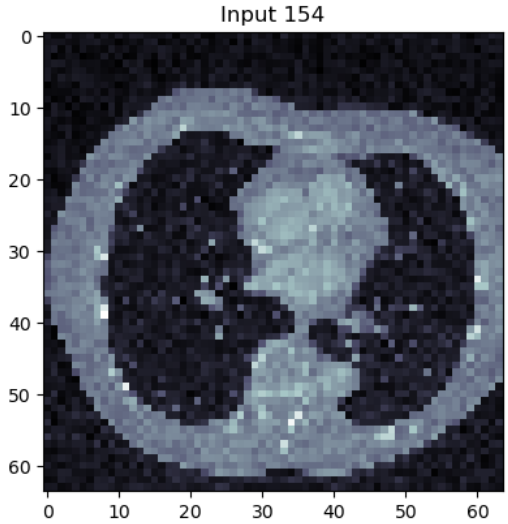

Mask 148
0
10
20
30
40
50
60
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00

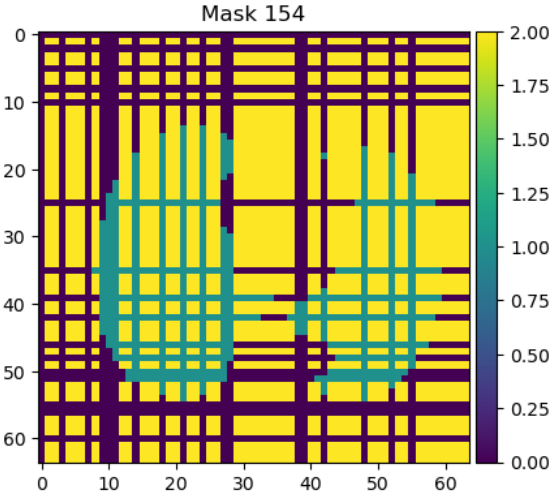

Input 150
0
10
20
30
40
50
60

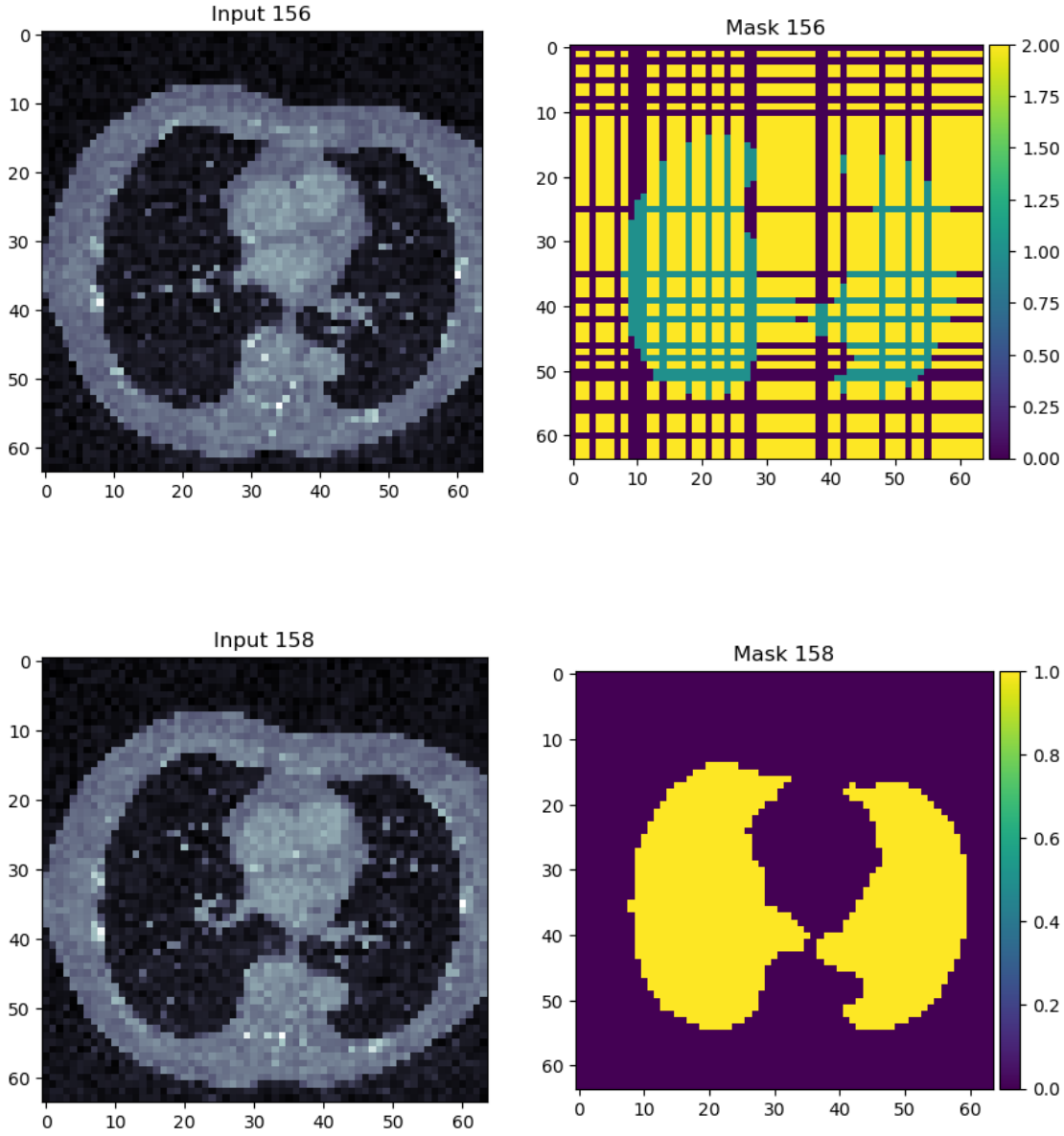

Mask 150
0
10
20
30
40
50
60
1.0
0.8
0.6
0.4
0.2
0.0


Input 152
0
10
20
30
40
50
60


Mask 152
0
10
20
30
40
50
60
1.0
0.8
0.6
0.4
0.2
0.0


Input 154
0
10
20
30
40
50
60


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.

[8]:
```
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 #
            x, y, z, none = np.shape(image)
            idx = int(np.random.choice(x - 64, 1))
            image_chunk = image[idx:idx+64, :, :]
            mask_chunk = mask[idx:idx+64, :, :]
            flat_mask_chunk = mask_chunk.flatten()
            weighted_mask = [1 if i < 2 else 0 for i in flat_mask_chunk]
            sample_weight = np.array(weighted_mask).reshape(slice_count,␣
 ↪slice_count, slice_count)
            # 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.

[9]:
```
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([16, 64, 64, 64, 1])
```

[9]: `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.

There are two use cases for the 3D U-Net model:

1. In a semi-automated setup, the user annotates a few slices in the volume to be segmented. The network learns from those sparse annotations and provides a dense 3D segmentation. This method assumes that the user needs a full segmentation of a small number of volumetric images, and does not have prior segmentations.
2. In a fully-automated setup, we assume that a representative, sparsely annotated training set exists. The network is trained with annotated slices from a representative training set and can be run on non-annotated volumes. Thus, trained on this existing dataset, the network densely segments new volumetric images. This setup assumes that the user wants to segment a large number of images recorded in a comparable setting and that there exists a representive training set.

Since we generate a training set of sparse slices from one set of images and using it to predict

another, never before seen slice, we are using the fully-automatic setup that is described in the paper.

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 each operation to its 3D counterpart! 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 should 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.

[10]:
```
from tensorflow.keras.layers import Conv3D, MaxPool3D, UpSampling3D, concatenate

class U_Net(tf.keras.Model):

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

        self.pool = MaxPool3D((2, 2, 2))
        self.conv1 = Conv3D(64, 3, activation = 'relu', padding = 'same')
        self.conv2 = Conv3D(128, 3, activation = 'relu', padding = 'same')
        self.conv3 = Conv3D(256, 3, activation = 'relu', padding = 'same')

        #upsampling
        self.up = UpSampling3D((2, 2, 2))
        self.conv4a = Conv3D(128, 2, activation = 'relu', padding = 'same')
        self.conv4b = Conv3D(128, 3, activation = 'relu', padding = 'same')
        self.conv5a = Conv3D(64, 2, activation = 'relu', padding = 'same')
        self.conv5b = Conv3D(64, 3, activation = 'relu', padding = 'same')
        self.final_conv = Conv3D(1, 1, activation = 'sigmoid', padding = 'same')

    def call(self, inputs):
        conv1 = self.conv1(inputs)
        pool1 = self.pool(conv1)
        conv2 = self.conv2(pool1)
        pool2 = self.pool(conv2)
        conv3 = self.conv3(pool2)
```

```
        up1 = self.up(conv3)
        conv4a = self.conv4a(up1)
        merge1 = concatenate([conv2,conv4a], axis = 4)
        conv4b = self.conv4b(merge1)
        up2 = self.up(conv4b)
        conv5a = self.conv5a(up2)
        merge2 = concatenate([conv1,conv5a], axis = 4)
        conv5b = self.conv5b(merge2)

        output = self.final_conv(conv5b)

        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, and `learning_rate` to be 3e-5. In our experiments, a model implementing the suggested architecture achieves a validation loss of just under 0.05. 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.

[11]:
```
STEPS_PER_EPOCH = 20
VALIDATION_STEPS = 20
EPOCHS = 20
LEARNING_RATE = 3e-5

# FILL IN CODE HERE #

optimizer = tf.keras.optimizers.Adam(learning_rate = LEARNING_RATE)
loss = tf.keras.losses.BinaryCrossentropy()
metrics = ["accuracy"]

model.compile(optimizer, loss, metrics)
model.fit(train_dataset, steps_per_epoch = STEPS_PER_EPOCH, epochs = EPOCHS,␣
 ↪validation_data = val_dataset, validation_steps= VALIDATION_STEPS)

# FILL IN CODE HERE #
```

```
Epoch 1/20
20/20 [==============================] - 150s 6s/step - loss: 0.3823 - accuracy:
0.5066 - val_loss: 0.3671 - val_accuracy: 0.4757
Epoch 2/20
20/20 [==============================] - 124s 6s/step - loss: 0.2847 - accuracy:
```

```
0.5080 - val_loss: 0.3458 - val_accuracy: 0.4846
Epoch 3/20
20/20 [==============================] - 124s 6s/step - loss: 0.2624 - accuracy:
0.5103 - val_loss: 0.3124 - val_accuracy: 0.5018
Epoch 4/20
20/20 [==============================] - 121s 6s/step - loss: 0.2363 - accuracy:
0.5076 - val_loss: 0.2897 - val_accuracy: 0.4848
Epoch 5/20
20/20 [==============================] - 118s 6s/step - loss: 0.1919 - accuracy:
0.5130 - val_loss: 0.2392 - val_accuracy: 0.4791
Epoch 6/20
20/20 [==============================] - 120s 6s/step - loss: 0.1308 - accuracy:
0.5065 - val_loss: 0.1614 - val_accuracy: 0.4824
Epoch 7/20
20/20 [==============================] - 119s 6s/step - loss: 0.0934 - accuracy:
0.5084 - val_loss: 0.1284 - val_accuracy: 0.4793
Epoch 8/20
20/20 [==============================] - 119s 6s/step - loss: 0.0871 - accuracy:
0.5058 - val_loss: 0.1200 - val_accuracy: 0.4800
Epoch 9/20
20/20 [==============================] - 118s 6s/step - loss: 0.0778 - accuracy:
0.5680 - val_loss: 0.1103 - val_accuracy: 0.5748
Epoch 10/20
20/20 [==============================] - 119s 6s/step - loss: 0.0655 - accuracy:
0.5843 - val_loss: 0.0630 - val_accuracy: 0.5825
Epoch 11/20
20/20 [==============================] - 121s 6s/step - loss: 0.0318 - accuracy:
0.5856 - val_loss: 0.0446 - val_accuracy: 0.5867
Epoch 12/20
20/20 [==============================] - 118s 6s/step - loss: 0.0246 - accuracy:
0.5852 - val_loss: 0.0415 - val_accuracy: 0.5850
Epoch 13/20
20/20 [==============================] - 120s 6s/step - loss: 0.0216 - accuracy:
0.5888 - val_loss: 0.0400 - val_accuracy: 0.5887
Epoch 14/20
20/20 [==============================] - 118s 6s/step - loss: 0.0206 - accuracy:
0.5917 - val_loss: 0.0386 - val_accuracy: 0.5881
Epoch 15/20
20/20 [==============================] - 120s 6s/step - loss: 0.0190 - accuracy:
0.5924 - val_loss: 0.0372 - val_accuracy: 0.5881
Epoch 16/20
20/20 [==============================] - 118s 6s/step - loss: 0.0192 - accuracy:
0.5935 - val_loss: 0.0455 - val_accuracy: 0.5890
Epoch 17/20
20/20 [==============================] - 118s 6s/step - loss: 0.0173 - accuracy:
0.5954 - val_loss: 0.0348 - val_accuracy: 0.5905
Epoch 18/20
20/20 [==============================] - 119s 6s/step - loss: 0.0171 - accuracy:
```

```
0.5923 - val_loss: 0.0374 - val_accuracy: 0.5889
Epoch 19/20
20/20 [==============================] - 120s 6s/step - loss: 0.0164 - accuracy:
0.5921 - val_loss: 0.0327 - val_accuracy: 0.5920
Epoch 20/20
20/20 [==============================] - 120s 6s/step - loss: 0.0165 - accuracy:
0.5911 - val_loss: 0.0372 - val_accuracy: 0.5885
```

[11]:
```
<keras.callbacks.History at 0x7f008eb0ce10>
```

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.

[16]:
```
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 #
    iou = np.sum(np.logical_and(target, prediction))/np.sum(np.
 ↪logical_or(target, prediction))
    # 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 zero-padding and reshaping operations in the `pad_and_reshape_image` function, which 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** and the padding should extend the image to the desired size.

[37]:
```
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␣
↪(new_dim, 64, 64, 1)
    """

    # FILL IN CODE HERE #
    x, y, z, not_needed = np.shape(image)
    if (new_dim - x)%2 != 0:
        to_add = (new_dim - x)//2
        npad = ((new_dim - x - to_add, new_dim - x - (to_add + 1)), (0, 0), (0,␣
↪0), (0, 0))
        padded_image = np.pad(image, pad_width=npad, mode='constant',␣
↪constant_values=0)
    else:
        to_add = (new_dim - x)//2
        npad = ((new_dim - x - to_add, new_dim - x - to_add + 1), (0, 0), (0,␣
↪0), (0, 0))
        padded_image = np.pad(image, pad_width=npad, mode='constant',␣
↪constant_values=0)
    # FILL IN CODE HERE #

    return padded_image.reshape(1, new_dim, y, z, not_needed)

padded_test_image = pad_and_reshape_image(test_img, 128)
print(padded_test_image.shape)  # (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`.
- Remember to add a new dimension to the padded image before axis 0 to account for batching.
- 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.

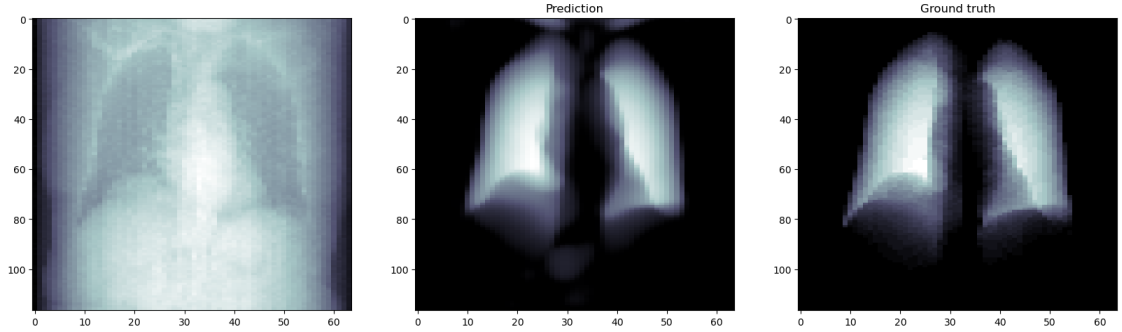
[56]:
```
# FILL IN CODE HERE #
scores = model.predict(padded_test_image)
x, y, z, not_needed = test_mask.shape
na_0, new_dim, na_1, na_2, na_3 = scores.shape
```

```
to_add = (new_dim - x)//2
scores = scores[:, to_add:x+to_add , :, :]
binarized_scores = scores > 0.5

IOU = compute_IoU(test_mask, binarized_scores)
print("our IOU is : {}".format(IOU))

# FILL IN CODE HERE #
```

```
our IOU is : 0.8674674058104918
```

Q1c.11: Finally, let's plot a sample prediction on the test dataset, with the 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.

[54]:
```
test_pred = scores.reshape(117, 64, 64)
test_img_sq = test_img.squeeze()      # (117, 64, 64)
test_mask_sq = test_mask.squeeze()    # (117, 64, 64)
test_pred = test_pred.squeeze()       # (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)
```



[ ]:

[ ]: