A1_part2_2d_lung_segmentation

October 19, 2022

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 to screen 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
from os import listdir
from os.path import isfile, join
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 IMAGE_DIR and MASK_DIR with the output paths.

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

IMAGE_DIR = "/home/jupyter/assign1/2d_images/"
MASK_DIR = "/home/jupyter/assign1/2d_masks/"
PROJECT_DIR = "home/jupyter/"

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

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).

[68]:
```
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)
        #images
        image = cv2.imread(image_path, cv2.IMREAD_UNCHANGED)
        image = image/image.max()
        image = cv2.resize(image, IMAGE_SIZE, interpolation = cv2.
 ↪INTER_LANCZOS4)
        images.append(image)
        #masks
        mask = cv2.imread(mask_path, cv2.IMREAD_UNCHANGED).astype('int16')
        mask = mask/mask.max()
        mask = cv2.resize(mask, IMAGE_SIZE, interpolation = cv2.INTER_NEAREST)
        masks.append(mask)

        # FILL IN CODE HERE #

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

    # FILL IN CODE HERE #
    split_1 = int(0.6 * len(all_images))
    split_2 = int(0.8 * len(all_images))
    train_images = all_images[:split_1]
    train_masks = all_masks[:split_1]
    val_images = all_images[split_1:split_2]
    val_masks = all_masks[split_1:split_2]
    test_images = all_images[split_2:]
    test_masks = all_masks[split_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 masks: ', train_masks.shape)
```

```
Train images:  (160, 128, 128, 1)
Train masks:  (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.

Data augmentations help expose the model to a larger variety of training samples by slightly modifying or warping the image via shift, rotations, zooms, etc. Data augmentation introduces greater diversity in the dataset, and helps reduce overfitting when training a machine learning model. It is possible for data augmentation to instead decrease the performance of the trained model on the test set. If the spatial orientation of the image was important for prediction power, then rotating and shifting the image would lead to underfitting. Additionally, if we zoomed into the image too much, we could lose the part of the image important for prediction, which would also lead to decreased performance.

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.

[76]:
```
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 #
    # Define `image_datagen` and `mask_datagen`
    data_gen_args = dict(
                      rotation_range=10,
                      width_shift_range=0.1,
                      height_shift_range=0.1,
                      zoom_range=0.1)
```

```
    image_datagen = ImageDataGenerator(**data_gen_args)
    mask_datagen = ImageDataGenerator(**data_gen_args)
    # 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)
img, mask = next(data_generator)
print(f"Image batch shape {img.shape}, mask batch shape {mask.shape}")
```

```
Image batch shape (16, 128, 128, 1), mask batch shape (16, 128, 128, 1)
```

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

[77]:
```
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, val_masks)).
 ↪batch(BATCH_SIZE)
test_dataset = tf.data.Dataset.from_tensor_slices((test_images, 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 masks.

[78]:
```
# FILL IN CODE HERE #
it = iter(train_dataset)
nrows, ncols = 4, 2
```

```
f, axs = plt.subplots(nrows, ncols, figsize=(5, 12))
for i in range(nrows):
    next_it = next(it)
    axs[i, 0].imshow(np.array(next_it)[0][i])
    axs[i, 1].imshow(np.array(next_it)[1][i])
    axs[i, 0].set(title='image')
    axs[i, 1].set(title='mask')
# FILL IN CODE HERE #
```

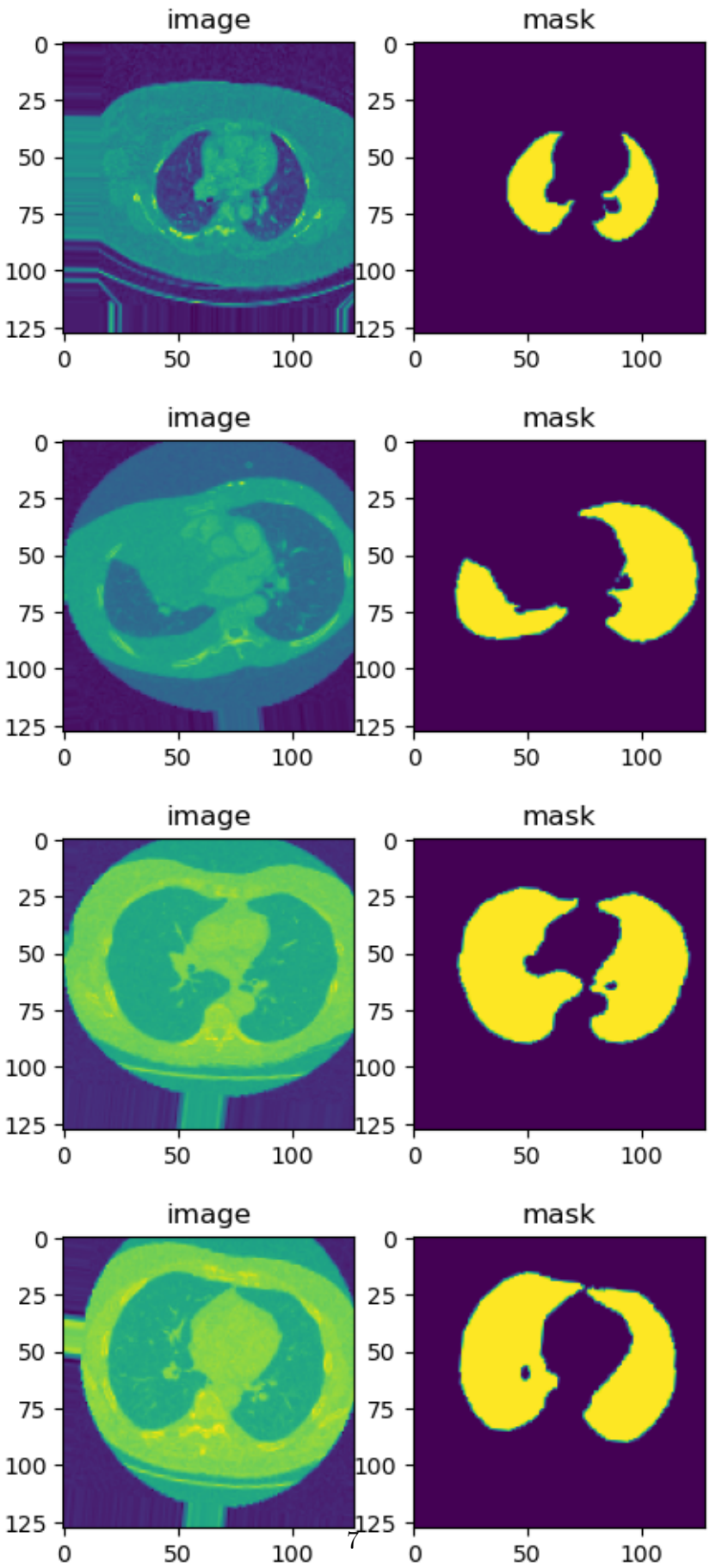


image
mask
0
25
50
75
100
125
0
50
100

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).

A U-Net is an architecture used for semantic segmentation. Its input is an image and its output is an image mask with width x height x #classes. A U-Net consists of two paths - an encoder network followed by a decoder network.

In the encoder (contracting) path, convolutional layers are followed by max pooling operations to make up a contracting block. The number of kernels in this path doubles after each block, so that the arcitecture can learn the complex structures more effectively. Additionally, the max pooling operation enables aggregation of increasingly more context (higher level features) as we travel down the contracting path.

In the decoder (expanding) path, upsampling of the feature map followed by a convolution (up-convolution) makes up the contracting block. The number of kernels in the contracting block are halved after each block, and the output of the previous block is concatenated with corresponding same-resolution feature map from the contracting path. The concatenations are utilized to combine high-level information with low-level (local) information. They add extra information on the decoder side of the network that may have been lost during downsampling. The up-convolutions are used to go from the global information encoded in the highest-level features back to the individual pixel predictions. At the end of the decoder path, we are able to more accurately predict classes for each pixel.

A U-Net is more suitable for segmentation tasks relative to a simple CNN. In a CNN, the image is converted into a vector which is used for classification problems. However, in a U-Net, an image is converted into a vector and then the same mapping is used to turn the vector back into an image. Thus, a CNN is used when we would like to classify the entire image as a class label. However, a U-Net is necessary if we would like to make per-pixel predictions, since the original image is reconstructed and each pixel has a predicted class label.

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: - **MaxPool/Upsampling steps**: 2 `MaxPool2D` layers on the left branch, and 2 `UpSampling2D` layers on the right branch (original paper architecture has 4 each). We used `size=(2, 2)` for both `MaxPool2D`

and `UpSampling2D`. - **Filters**: The U-Net 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. We used dropout rate 0.5. - **Final layer**: A `Conv2d` layer that has an output with 1 channel restricted to the range [0,1].

[79]:
```
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 #
        #downsampling
        self.pool = MaxPool2D((2, 2))
        self.conv1 = Conv2D(64, 3, activation = 'relu', padding = 'same')
        self.conv2 = Conv2D(128, 3, activation = 'relu', padding = 'same')
        self.conv3 = Conv2D(256, 3, activation = 'relu', padding = 'same')

        #upsampling
        self.up = UpSampling2D((2, 2))
        self.conv4a = Conv2D(128, 2, activation = 'relu', padding = 'same')
        self.conv4b = Conv2D(128, 3, activation = 'relu', padding = 'same')
        self.conv5a = Conv2D(64, 2, activation = 'relu', padding = 'same')
        self.conv5b = Conv2D(64, 3, activation = 'relu', padding = 'same')
        self.drop = Dropout(0.5)
        self.final_conv = Conv2D(1, 1, activation = 'sigmoid', padding = 'same')

        # FILL IN CODE HERE #

    def call(self, inputs):
        # FILL IN CODE HERE #
        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 = 3)
        conv4b = self.conv4b(merge1)
```

```
        up2 = self.up(conv4b)
        conv5a = self.conv5a(up2)
        merge2 = concatenate([conv1,conv5a], axis = 3)
        conv5b = self.conv5b(merge2)

        drop = self.drop(conv5b)
        output = self.final_conv(drop)

        # 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.)

The most appropriate loss function for this task is the binary cross entropy loss. Cross-entropy loss measures the performance of a classification model whose output is a probability value between 0 and 1. Cross-entropy loss increases as the predicted probability diverges from the actual label. For example, predicting a probability of 0.05 for an actual observation pixel of 1 would result in a high loss. Since we are trying to make per pixel predictions between binary classes, this loss is appropriate.

[80]:
```
optimizer = tf.keras.optimizers.Adam()
loss = tf.keras.losses.BinaryCrossentropy()
metrics = ["accuracy"]

# FILL IN CODE HERE #
model.compile(optimizer, loss, metrics)
# 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?

Decreasing the learning rate during training can be very useful when training a complex neural network. This method allows for large weight changes at the beginning of the learning process and

smaller changes towards the end, when you are looking to find-tune your model. At the beginning of training, larger weight changes are beneficial since they lead you towards an optimal solution more rapidly. However, once the weights are near optimal, a large learning rate change can cause large weight changes, leading to instability. Thus, decreasing the learning rate during training ensures that there are smaller weight changes as you get closer and closer to the optimal solution. This allows you to fine tune your model and avoid divergence.

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.

[74]:
```
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 #
    if epoch < 20:
        return 0.001
    else:
        return 0.001 * 0.1**(epoch - 20)

    # 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()`. To enable the learning rate schedule we just implemented, pass it as a callback to `model.fit`.

[81]:
```
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,
    callbacks = [lr_scheduler],
    steps_per_epoch = steps_per_epoch,
    validation_data = val_dataset
    # FILL IN CODE HERE #
)
```

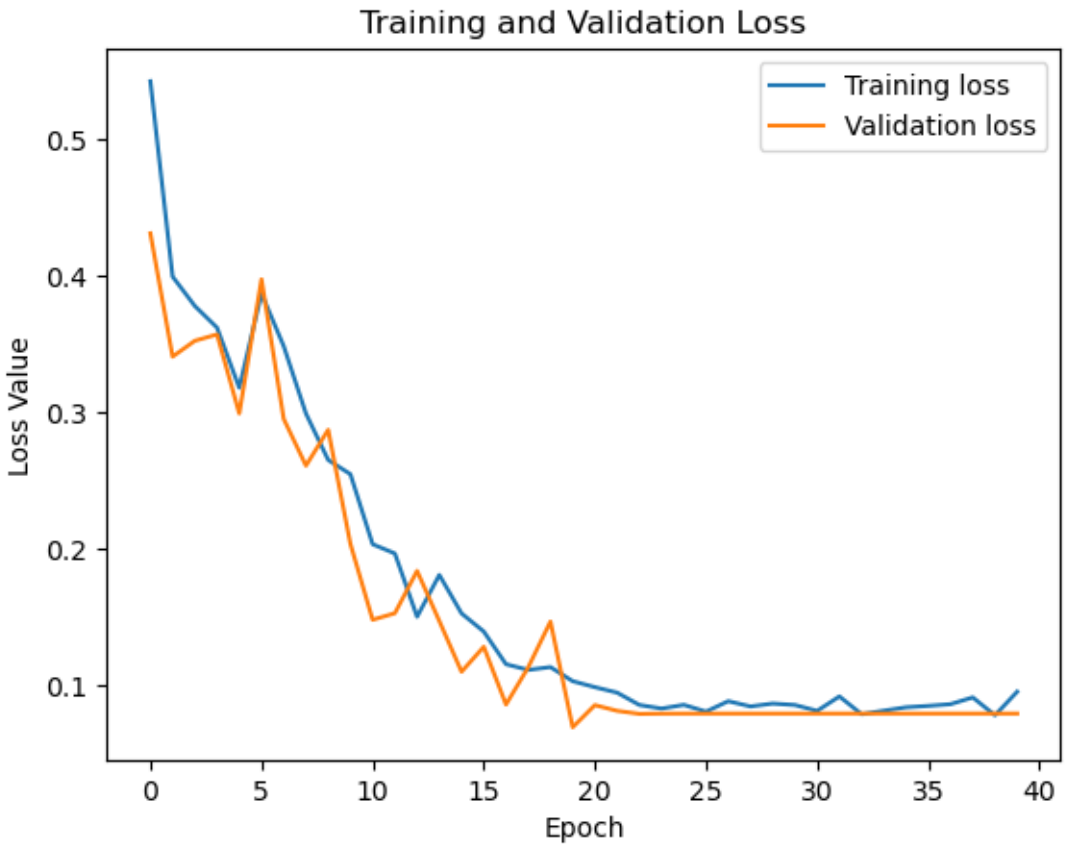
```
Epoch 1/40
10/10 [==============================] - 2s 173ms/step - loss: 0.5430 -
accuracy: 0.7152 - val_loss: 0.4314 - val_accuracy: 0.7755
Epoch 2/40
10/10 [==============================] - 2s 157ms/step - loss: 0.3997 -
accuracy: 0.7485 - val_loss: 0.3410 - val_accuracy: 0.7755
Epoch 3/40
10/10 [==============================] - 2s 157ms/step - loss: 0.3781 -
accuracy: 0.7415 - val_loss: 0.3526 - val_accuracy: 0.7755
Epoch 4/40
10/10 [==============================] - 2s 157ms/step - loss: 0.3623 -
accuracy: 0.7406 - val_loss: 0.3573 - val_accuracy: 0.7755
Epoch 5/40
10/10 [==============================] - 2s 158ms/step - loss: 0.3181 -
accuracy: 0.7517 - val_loss: 0.2993 - val_accuracy: 0.7802
Epoch 6/40
10/10 [==============================] - 2s 156ms/step - loss: 0.3874 -
accuracy: 0.7652 - val_loss: 0.3977 - val_accuracy: 0.7755
Epoch 7/40
10/10 [==============================] - 2s 157ms/step - loss: 0.3486 -
accuracy: 0.7485 - val_loss: 0.2953 - val_accuracy: 0.7807
Epoch 8/40
10/10 [==============================] - 2s 158ms/step - loss: 0.2993 -
accuracy: 0.8062 - val_loss: 0.2611 - val_accuracy: 0.9085
Epoch 9/40
10/10 [==============================] - 2s 157ms/step - loss: 0.2652 -
accuracy: 0.8677 - val_loss: 0.2873 - val_accuracy: 0.8583
Epoch 10/40
10/10 [==============================] - 2s 158ms/step - loss: 0.2548 -
accuracy: 0.8771 - val_loss: 0.2037 - val_accuracy: 0.9224
Epoch 11/40
10/10 [==============================] - 2s 159ms/step - loss: 0.2034 -
accuracy: 0.9017 - val_loss: 0.1480 - val_accuracy: 0.9417
Epoch 12/40
10/10 [==============================] - 2s 160ms/step - loss: 0.1967 -
accuracy: 0.9005 - val_loss: 0.1528 - val_accuracy: 0.9445
Epoch 13/40
10/10 [==============================] - 2s 158ms/step - loss: 0.1503 -
accuracy: 0.9278 - val_loss: 0.1839 - val_accuracy: 0.9215
Epoch 14/40
10/10 [==============================] - 2s 158ms/step - loss: 0.1809 -
accuracy: 0.9120 - val_loss: 0.1471 - val_accuracy: 0.9359
Epoch 15/40
10/10 [==============================] - 2s 159ms/step - loss: 0.1527 -
accuracy: 0.9215 - val_loss: 0.1099 - val_accuracy: 0.9584
Epoch 16/40
10/10 [==============================] - 2s 159ms/step - loss: 0.1394 -
accuracy: 0.9308 - val_loss: 0.1282 - val_accuracy: 0.9500
```

```
Epoch 17/40
10/10 [==============================] - 2s 159ms/step - loss: 0.1154 -
accuracy: 0.9374 - val_loss: 0.0859 - val_accuracy: 0.9694
Epoch 18/40
10/10 [==============================] - 2s 159ms/step - loss: 0.1113 -
accuracy: 0.9389 - val_loss: 0.1132 - val_accuracy: 0.9562
Epoch 19/40
10/10 [==============================] - 2s 159ms/step - loss: 0.1133 -
accuracy: 0.9426 - val_loss: 0.1467 - val_accuracy: 0.9519
Epoch 20/40
10/10 [==============================] - 2s 159ms/step - loss: 0.1030 -
accuracy: 0.9399 - val_loss: 0.0692 - val_accuracy: 0.9767
Epoch 21/40
10/10 [==============================] - 2s 160ms/step - loss: 0.0986 -
accuracy: 0.9459 - val_loss: 0.0854 - val_accuracy: 0.9703
Epoch 22/40
10/10 [==============================] - 2s 161ms/step - loss: 0.0946 -
accuracy: 0.9497 - val_loss: 0.0813 - val_accuracy: 0.9693
Epoch 23/40
10/10 [==============================] - 2s 160ms/step - loss: 0.0856 -
accuracy: 0.9517 - val_loss: 0.0790 - val_accuracy: 0.9705
Epoch 24/40
10/10 [==============================] - 2s 172ms/step - loss: 0.0829 -
accuracy: 0.9526 - val_loss: 0.0792 - val_accuracy: 0.9704
Epoch 25/40
10/10 [==============================] - 2s 166ms/step - loss: 0.0857 -
accuracy: 0.9517 - val_loss: 0.0793 - val_accuracy: 0.9703
Epoch 26/40
10/10 [==============================] - 2s 160ms/step - loss: 0.0807 -
accuracy: 0.9532 - val_loss: 0.0793 - val_accuracy: 0.9703
Epoch 27/40
10/10 [==============================] - 2s 160ms/step - loss: 0.0883 -
accuracy: 0.9509 - val_loss: 0.0793 - val_accuracy: 0.9703
Epoch 28/40
10/10 [==============================] - 2s 159ms/step - loss: 0.0845 -
accuracy: 0.9521 - val_loss: 0.0793 - val_accuracy: 0.9703
Epoch 29/40
10/10 [==============================] - 2s 160ms/step - loss: 0.0866 -
accuracy: 0.9509 - val_loss: 0.0793 - val_accuracy: 0.9703
Epoch 30/40
10/10 [==============================] - 2s 160ms/step - loss: 0.0855 -
accuracy: 0.9530 - val_loss: 0.0793 - val_accuracy: 0.9703
Epoch 31/40
10/10 [==============================] - 2s 160ms/step - loss: 0.0812 -
accuracy: 0.9531 - val_loss: 0.0793 - val_accuracy: 0.9703
Epoch 32/40
10/10 [==============================] - 2s 159ms/step - loss: 0.0918 -
accuracy: 0.9509 - val_loss: 0.0793 - val_accuracy: 0.9703
```

```
Epoch 33/40
10/10 [==============================] - 2s 160ms/step - loss: 0.0788 -
accuracy: 0.9543 - val_loss: 0.0793 - val_accuracy: 0.9703
Epoch 34/40
10/10 [==============================] - 2s 159ms/step - loss: 0.0814 -
accuracy: 0.9520 - val_loss: 0.0793 - val_accuracy: 0.9703
Epoch 35/40
10/10 [==============================] - 2s 158ms/step - loss: 0.0839 -
accuracy: 0.9529 - val_loss: 0.0793 - val_accuracy: 0.9703
Epoch 36/40
10/10 [==============================] - 2s 158ms/step - loss: 0.0849 -
accuracy: 0.9512 - val_loss: 0.0793 - val_accuracy: 0.9703
Epoch 37/40
10/10 [==============================] - 2s 158ms/step - loss: 0.0861 -
accuracy: 0.9526 - val_loss: 0.0793 - val_accuracy: 0.9703
Epoch 38/40
10/10 [==============================] - 2s 157ms/step - loss: 0.0911 -
accuracy: 0.9494 - val_loss: 0.0793 - val_accuracy: 0.9703
Epoch 39/40
10/10 [==============================] - 2s 157ms/step - loss: 0.0780 -
accuracy: 0.9544 - val_loss: 0.0793 - val_accuracy: 0.9703
Epoch 40/40
10/10 [==============================] - 2s 158ms/step - loss: 0.0953 -
accuracy: 0.9491 - val_loss: 0.0793 - val_accuracy: 0.9703
```

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

[82]:
```
plt.figure()
plt.plot(range(EPOCHS), hist.history['loss'], label='Training 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.

[83]:
```
model.evaluate(test_dataset)
```

```
4/4 [==============================] - 1s 269ms/step - loss: 0.0718 - accuracy:
0.9783
```

[83]:
```
[0.07180506736040115, 0.9783042669296265]
```

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.

No, I don't think we are overfitting. The validation loss does not increase towards the end while the training loss continues to decrease. They are both relatively constant at around epoch 20, showing that the model isn't improving much after epoch 20 but it isn't overfitting.

Methods to avoid overfitting:

1) using dropout, a technique where randomly selected neurons are ignored during training.

2) adding more training examples through increased data collection or data augmentation.

3) applying regularization, which adds a cost to the loss function for large weights.

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.

Intersection over Union is defined as the number of pixels in common in both the target and prediction masks divided total number of pixels in the union of both maps. IoU is a useful metric for segmentation tasks since it measures the per pixel degree of similarity between a preidction mask and a target mask. IOU is useful for segmentation tasks since it is less sensitive to class imbalance in the way that pixel accuracy is. Pixel accuracy could still achieve a very high score if you have lots of background and it predicts zero for every pixel.

[132]:
```
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 #
    iou = np.sum(np.logical_and(target, prediction))/np.sum(np.
 ↪logical_or(target, prediction))
    # 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.

[183]:
```
# FILL IN CODE HERE #
total_IOU = []
scores = model.predict(test_dataset)
binarized_scores = scores > 0.1

for i in np.arange(len(test_masks)):
    total_IOU.append(compute_IoU(test_masks[i], binarized_scores[i]))

mean_IOU = np.mean(total_IOU)
mean_IOU
# FILL IN CODE HERE #
```

[183]:
```
0.8225368742494685
```

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].

[193]:
```
# FILL IN CODE HERE #
IOU_85 = []
IOU_90 = []
IOU_95 = []

for i in np.arange(len(test_masks)):
    IOU = compute_IoU(test_masks[i], binarized_scores[i])
    if (IOU >= 0.85 and IOU < 0.9):
        IOU_85.append(i)
    elif (IOU >= 0.9 and IOU < 0.95):
        IOU_90.append(i)
    elif (IOU >= 0.95 and IOU <= 1.0):
        IOU_95.append(i)

pics = [IOU_85[0], IOU_90[0], IOU_90[1], IOU_95[0]]

nrows, ncols  = 4,3
f, axs = plt.subplots(nrows, ncols, figsize=(10, 15))

for i in np.arange(nrows):
    axs[i, 0].imshow(test_images[pics[i]])
    axs[i, 1].imshow(test_masks[pics[i]])
    axs[i, 2].imshow(binarized_scores[pics[i]])
    axs[i, 0].set(title='image')
    axs[i, 1].set(title='mask')
    axs[i, 2].set(title='prediction')



# FILL IN CODE HERE #
```

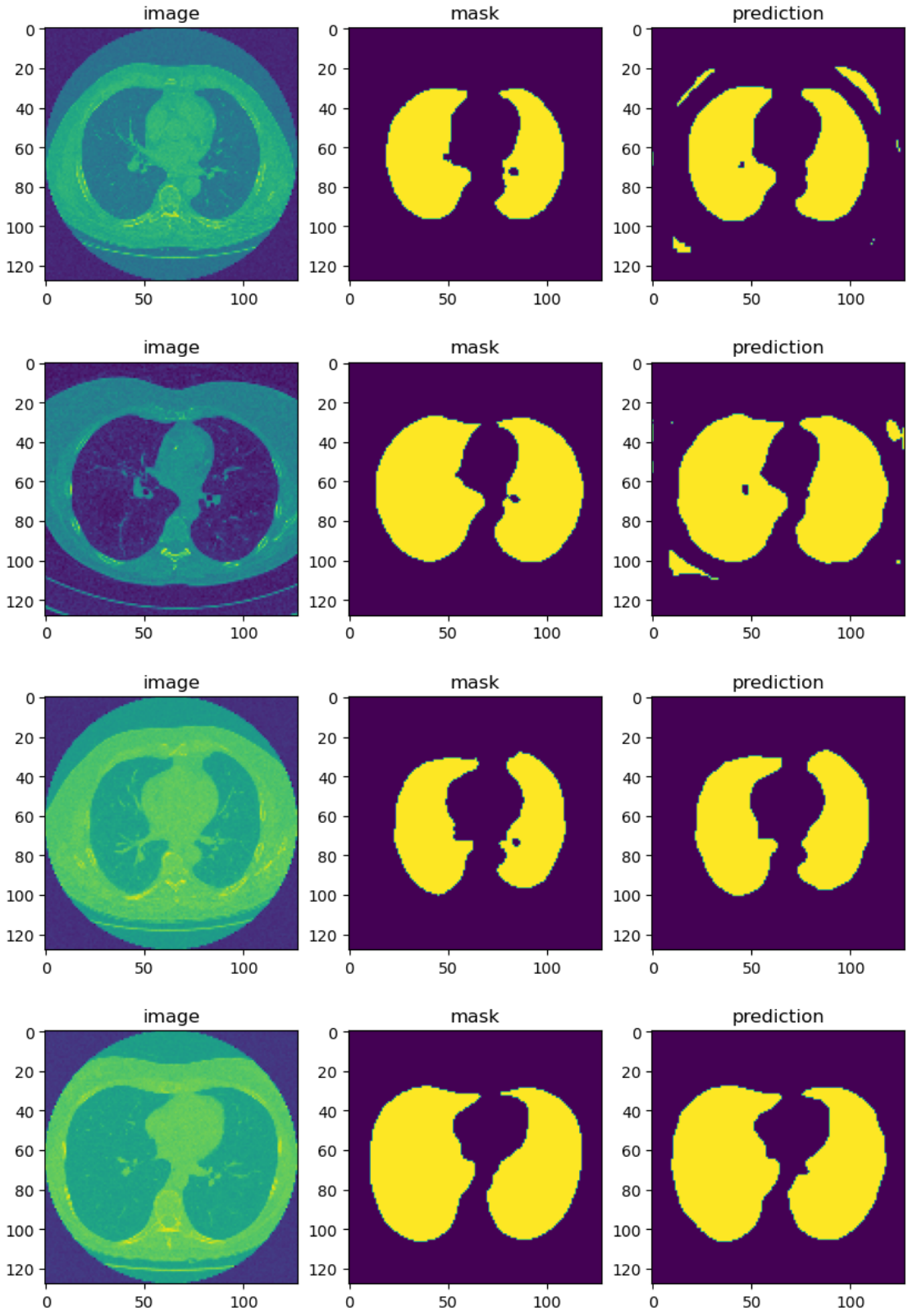


image
mask
prediction
0
20
40
60
80
100
120
0
50
100

[ ]:

[ ]: