CSE472 - Computer Vision
Programming Assignment 3

Student name: Nguyen Minh Duc
Student ID: 20202026

1 Problem 1

In this problem, we need to modify the `main.py` file and run the `Tester` class.

1.1 Download data and code

To begin with, I downloaded all of the required files on google drive and put them on my personal computer as follows

```
~/unist/cse472/assignment3/
├── data/
│   ├── test_img/
│   ├── test_label/
│   ├── train_img/
│   └── train_label/
├── main.py
└── model.py
```

1.2 Modify the data path

If one just runs the `main.py` file without any modification, errors will occur since the data path in `Tester` class is not correct. Thus, they need to be properly handled by modifying the path according to the directory tree above.

```python
class Tester(object):
    def __init__(self, batch_size):
        # ...
        # This part is modified
        dataset = Dataset(img_path="data/test_img", label_path="data/test_label",
    method='test')
        self.dataloader = torch.utils.data.DataLoader(dataset=dataset,
                                                      batch_size=self.batch_size,
                                                      shuffle=True,
                                                      num_workers=2,
                                                      drop_last=False)
        print("Testing...")

    def test(self):
        make_folder("test_mask", '')
        make_folder("test_color_mask", '')
        self.model.eval()
        for i, data in enumerate(self.dataloader):
            imgs = data[0].cuda()
            labels_predict = self.model(imgs)
            labels_predict_plain = generate_label_plain(labels_predict, 512)
            # The rest is the same ...

```

After modifying the code, `main.py` can now run normally.

1.3 Visualize the segmentation

After running `main.py`, I obtained two directories, each of which contained 500 images. As instructed in the Programming Assignment 3 handout, we need to visualize segmentation masks for the 6 mentioned images. They are as follows

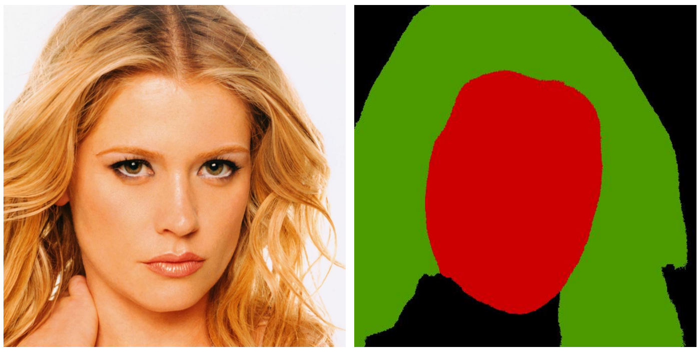

Figure 1: Image 0.jpg

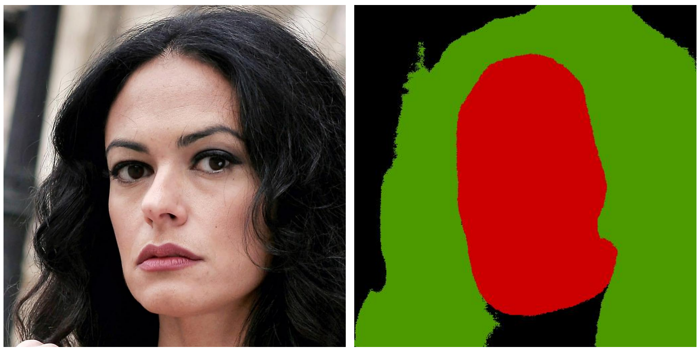

Figure 2: Image 10.jpg

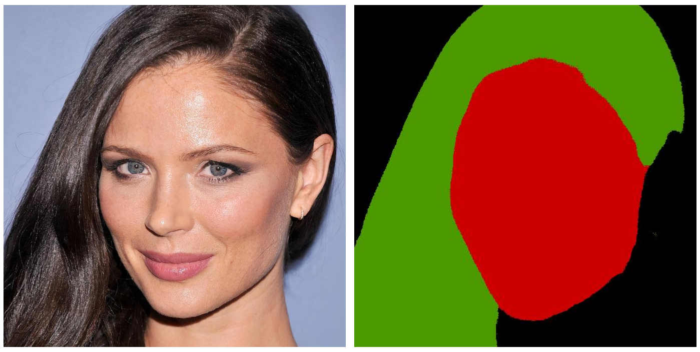

Figure 3: Image 100.jpg

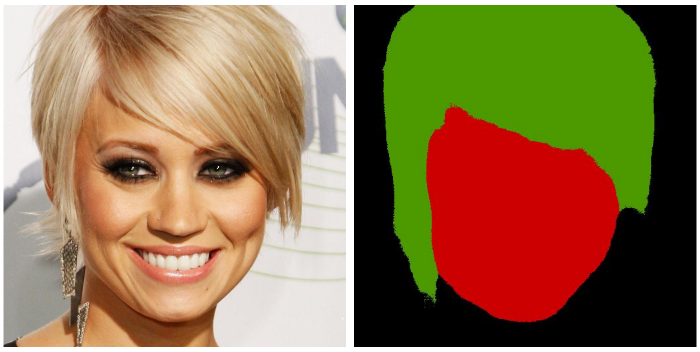

Figure 4: Image 103.jpg

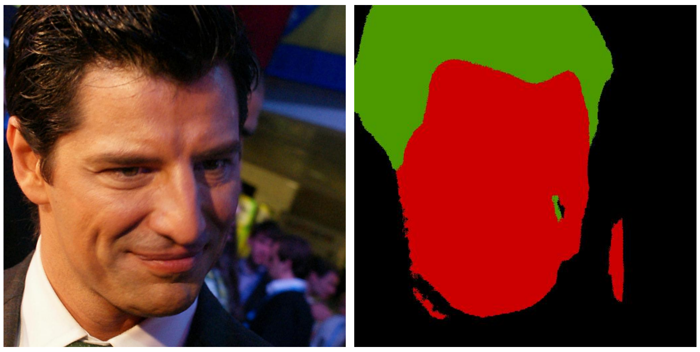

Figure 5: Image 109.jpg

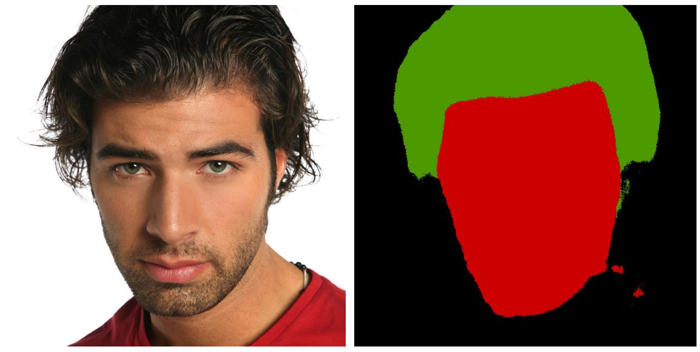

Figure 6: Image 162.jpg

The pre-trained weights yield a pretty good segmentation mask estimation. It could provide near-perfect estimations for images 0, 100, and 103. However, for the remaining three, the estimations seem to wrongly classify the noises and the background as facial landmarks. In the next problem, we will try to train our own segmentation mask estimator and hopefully will obtain better results.

2 Problem 2

In this problem, we are asked to implement the training process for the Segmentation model. Below is the detailed implementation. To begin with, we add several things to the constructor. Loss function, loading pre-trained weights, and loading test set are added to support the training process.

```python
class Trainer(object):
    def __init__(self, epochs, batch_size, lr):
        # ...
        self.dataloader = torch.utils.data.DataLoader(dataset=dataset,
                                                      batch_size=self.batch_size,
                                                      shuffle=True,
                                                      num_workers=2,
                                                      drop_last=False)

        # Added code
        # Create criteria, which is Cross Entropy Loss function
        self.loss_fn = nn.CrossEntropyLoss()
        # Load the pre-trained weights
        weight_PATH = 'pretrained_weight.pth'
        self.model.load_state_dict(torch.load(weight_PATH))
        # Array to save the training and testing loss values
        self.plot_train = []
        self.plot_test = []
        # Load test set
        testset = Dataset(img_path="data/test_img", label_path="data/test_label",
    method='test')
        self.testloader = torch.utils.data.DataLoader(dataset=testset,
                                                      batch_size=self.batch_size,
                                                      shuffle=True,
                                                      num_workers=2,
                                                      drop_last=False)
    # ...

```

Secondly, two more methods are added: `train_loop` and `test_loop`. The `train_loop` method is called when the model is learning, and `test_loop` is called when we want to test the performance of our model at a given epoch.

```python
class Trainer(object):
    # ...
    def train_loop(self):
        train_loss = 0
        for sample in tqdm(self.dataloader):
            # Bring data to gpu
            x, y_true, _ = sample
            x = x.cuda()
            y_true = y_true.squeeze().cuda() * 255
            # Get segmentation
            self.optimizer.zero_grad()
            y_pred = self.model(x)
            # y_pred = F.softmax(y_pred, dim=1)
            # Calculate the loss
            loss = self.loss_fn(y_pred, y_true.long())
            # Update the weights
            loss.backward()
            self.optimizer.step()
            # Adding up the total loss
            train_loss += loss.item()
        # Return the average loss
        return train_loss / len(self.dataloader)

    def test_loop(self):
        test_loss = 0
        for sample in tqdm(self.testloader):
            # Bring data to gpu
            x, y_true, _ = sample
            x = x.cuda()
            y_true = y_true.squeeze().cuda() * 255
            # Get segmentation
            y_pred = self.model(x)
            # y_pred = F.softmax(y_pred, dim=1)
            # Calculate the loss and add up the total loss
            loss = self.loss_fn(y_pred, y_true.long())
```

```
                test_loss += loss.item()
            # Return the average loss
            return test_loss / len(self.testloader)
    # ...

```

Finally, we can now start to implement the required `train` method by calling the above two methods. During the training process, the learning rate is reduced by a factor of 0.5 for every 10 epochs.

```
class Trainer(object):
    # ...
    def train(self):
        print("-" * 20, "Start training", "-" * 20)
        for epoch in range(self.epochs):
            print(f"Epoch {epoch + 1}/{self.epochs}: ")
            # Learning rate scheduler
            if epoch > 1 and epoch % 10 == 1:
                self.learning_rate *= 0.5
                self.optimizer = torch.optim.Adam(self.model.parameters(), self.
    learning_rate)
            # Train mode
            self.model.train()
            train_loss = self.train_loop()
            self.plot_train.append(train_loss)
            print(f"Average training loss: {train_loss:.6f}")
            # Test mode
            self.model.eval()
            with torch.no_grad():
                test_loss = self.test_loop()
                self.plot_test.append(test_loss)
                print(f"Average testing loss: {test_loss:.6f}")
            print("-" * 50)
    # ...

```

3 Problem 3

In this problem, we need to train our segmentation mask estimator using the code in problem 2. After trying a lot of configurations, we came down to the following choice of hyperparameters since they provide the best performance.

```
epochs = 20
lr = 5e-4
batch_size = 16

```

Figure 7 provides the behavior of the loss function during the training process. The average loss value on the test data set after training is 0.155564. We observe that the test loss starts to converge after around 15 epochs, so 20-epoch was chosen.
The visualization of our estimator's output could be found in Figure 8 to 13. It could still provide near-perfect estimations for images 0, 100, and 103. But now, image 10 and 162 is much smoother compared to the pre-trained one. Unfortunately, the estimator still confuses the background with hair and facial landmarks in image 109 due to the complex background that has other people. However, the resulting segmentation no longer estimates some hair in the middle of the face like the pre-trained one. The fine-tuned version can now recognize the ear feature, which the previous version does not.

4 Appendix

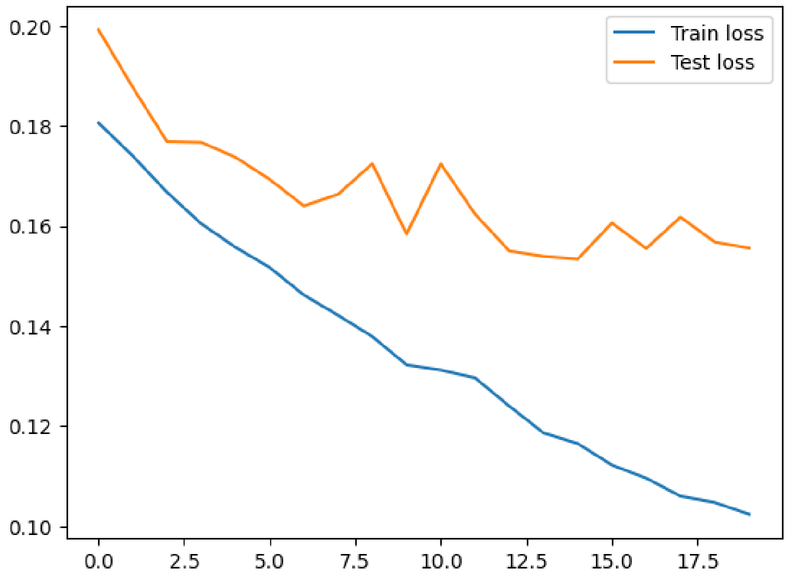

Figure 7: Loss values of train loss and test loss during the training process

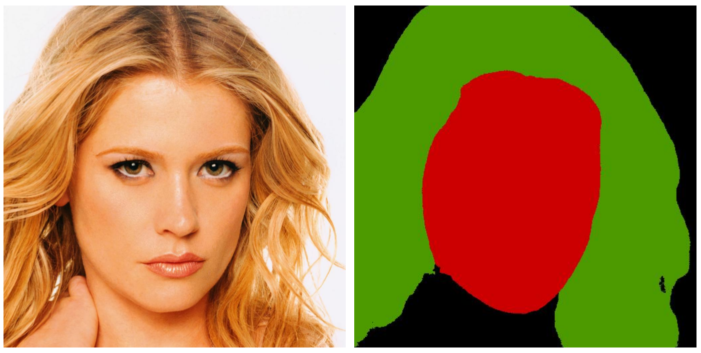

Figure 8: Image 0.jpg after tuning

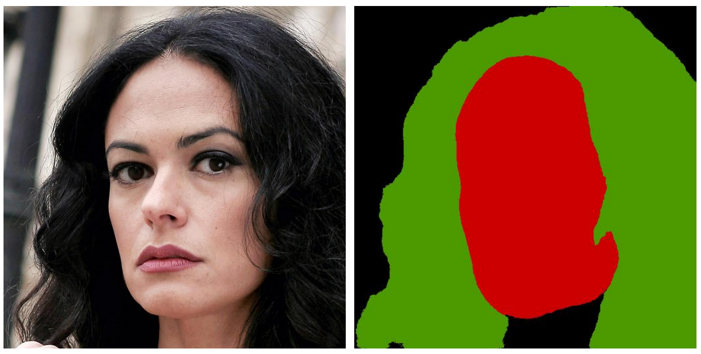

Figure 9: Image 10.jpg after tuning

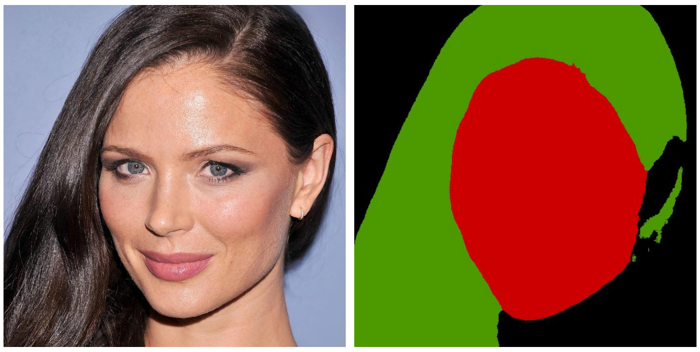

Figure 10: Image 100.jpg after tuning

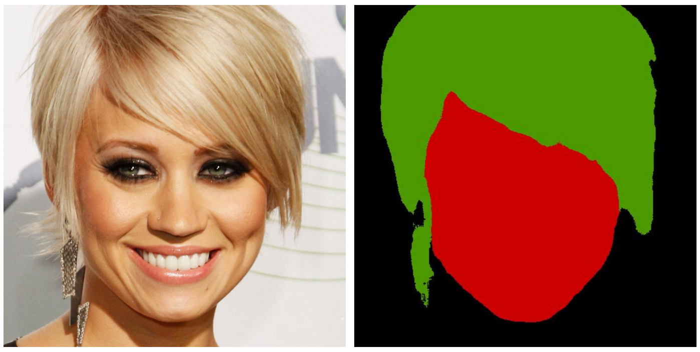

Figure 11: Image 103.jpg after tuning

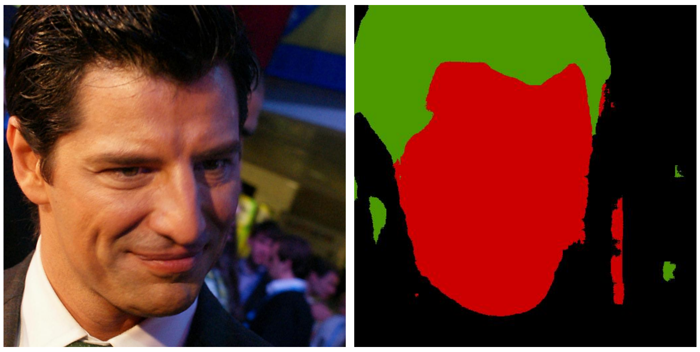

Figure 12: Image 109.jpg after tuning

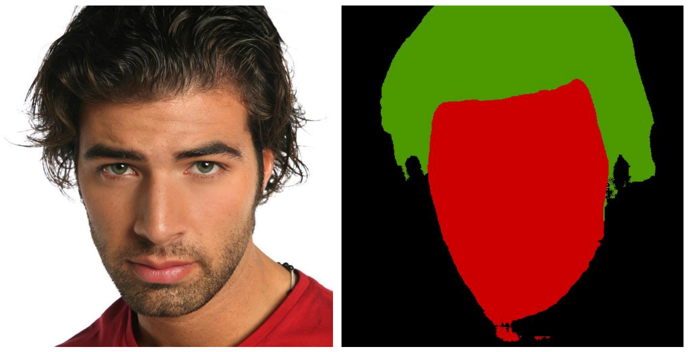

Figure 13: Image 162.jpg after tuning