3. Facial Keypoint Detection, Complete Pipeline

May 17, 2020

0.1 Face and Facial Keypoint detection

After you've trained a neural network to detect facial keypoints, you can then apply this network to *any* image that includes faces. The neural network expects a Tensor of a certain size as input and, so, to detect any face, you'll first have to do some pre-processing.

1. Detect all the faces in an image using a face detector (we'll be using a Haar Cascade detector in this notebook).
2. Pre-process those face images so that they are grayscale, and transformed to a Tensor of the input size that your net expects. This step will be similar to the `data_transform` you created and applied in Notebook 2, whose job was tp rescale, normalize, and turn any iimage into a Tensor to be accepted as input to your CNN.
3. Use your trained model to detect facial keypoints on the image.

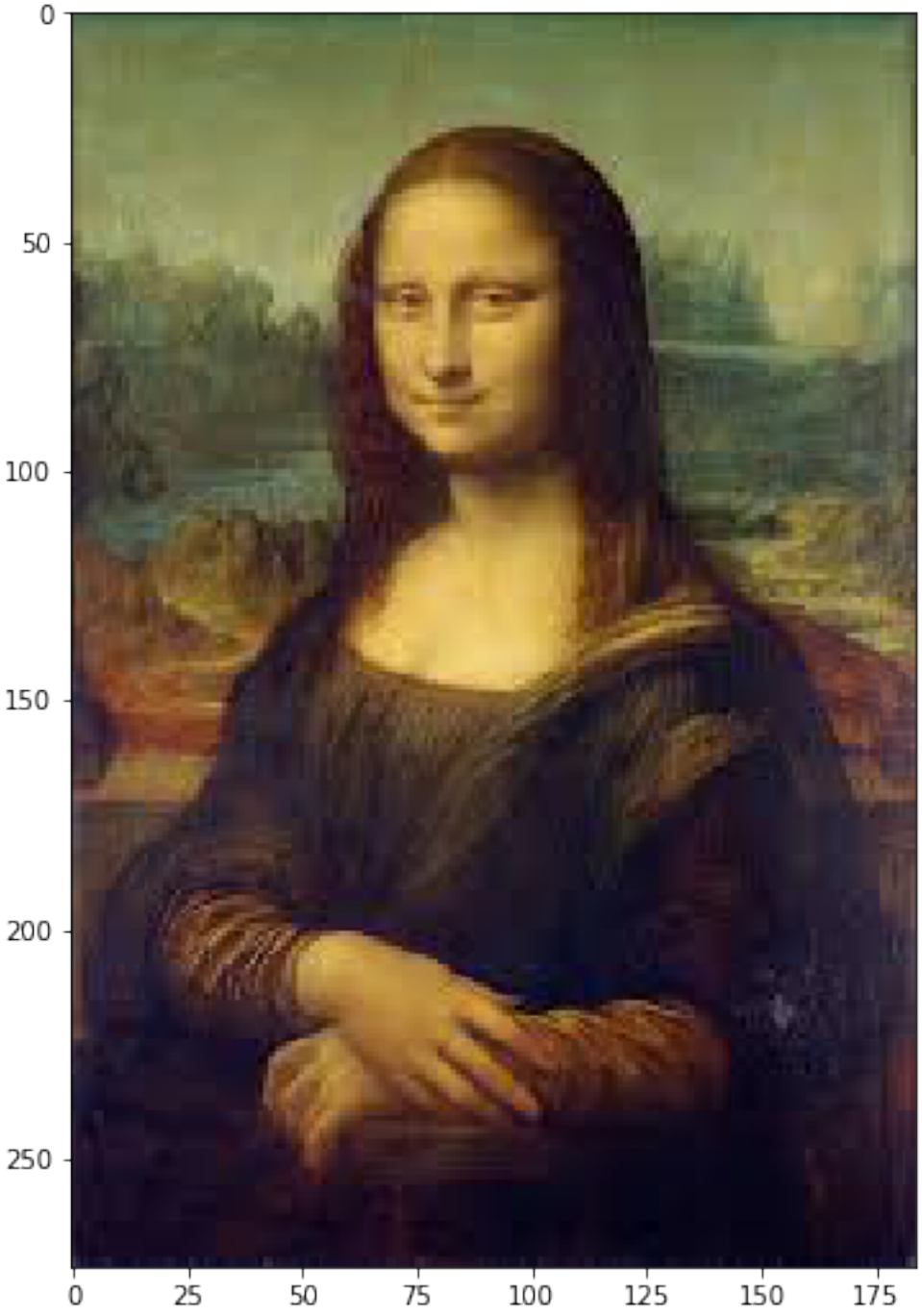
In the next python cell we load in required libraries for this section of the project.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib.image as mpimg
        %matplotlib inline
```

Select an image Select an image to perform facial keypoint detection on; you can select any image of faces in the `images/` directory.

```
In [2]: import cv2
        # load in color image for face detection
        image = cv2.imread('images/mona_lisa.jpg')

        # switch red and blue color channels
        # --> by default OpenCV assumes BLUE comes first, not RED as in many images
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

        # plot the image
        fig = plt.figure(figsize=(9,9))
        plt.imshow(image)
```

Out[2]: <matplotlib.image.AxesImage at 0x7efcdcfc72b0>



0.2 Detect all faces in an image

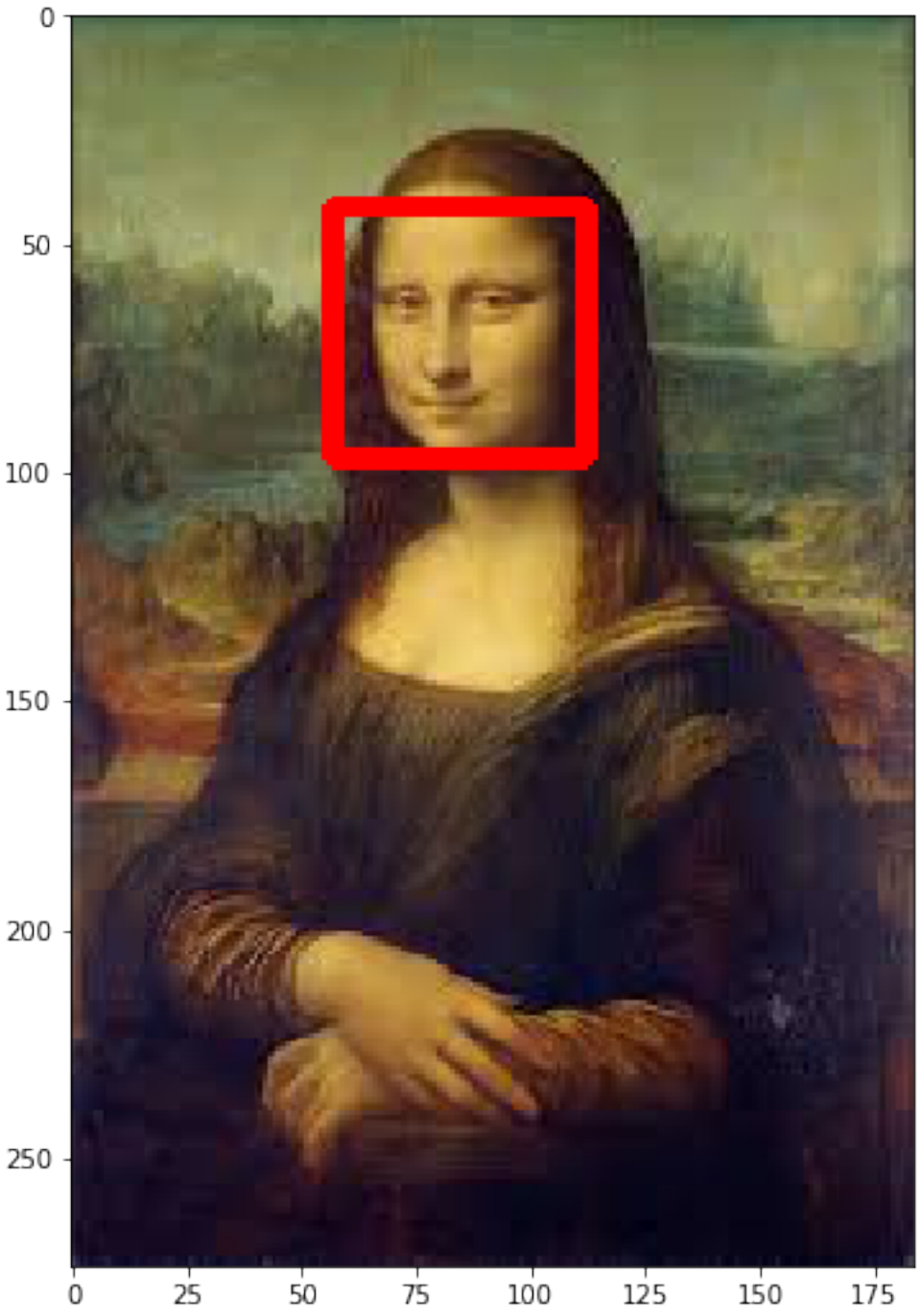
Next, you'll use one of OpenCV's pre-trained Haar Cascade classifiers, all of which can be found in the `detector_architectures/` directory, to find any faces in your selected image.

In the code below, we loop over each face in the original image and draw a red square on each face (in a copy of the original image, so as not to modify the original). You can even add eye detections as an *optional* exercise in using Haar detectors.

An example of face detection on a variety of images is shown below.

In [3]:
```
# load in a haar cascade classifier for detecting frontal faces
face_cascade = cv2.CascadeClassifier('detector_architectures/haarcascade_frontalface_def

# run the detector
# the output here is an array of detections; the corners of each detection box
# if necessary, modify these parameters until you successfully identify every face in a
faces = face_cascade.detectMultiScale(image, 1.2, 2)

# make a copy of the original image to plot detections on
image_with_detections = image.copy()

# loop over the detected faces, mark the image where each face is found
for (x,y,w,h) in faces:
    # draw a rectangle around each detected face
    # you may also need to change the width of the rectangle drawn depending on image re
    cv2.rectangle(image_with_detections,(x,y),(x+w,y+h),(255,0,0),3)

fig = plt.figure(figsize=(9,9))

plt.imshow(image_with_detections)
```

Out[3]:
```
<matplotlib.image.AxesImage at 0x7efcdcf6b4e0>
```



0.3 Loading in a trained model

Once you have an image to work with (and, again, you can select any image of faces in the `images/` directory), the next step is to pre-process that image and feed it into your CNN facial keypoint detector.

First, load your best model by its filename.

In [4]:
```
import torch
from models import Net

net = Net()

## TODO: load the best saved model parameters (by your path name)
## You'll need to un-comment the line below and add the correct name for *your* saved mo
net.load_state_dict(torch.load('saved_models/deep_keypoints_model.pt'))


## print out your net and prepare it for testing (uncomment the line below)
net.eval()
```

```
/home/workspace/models.py:46: UserWarning: nn.init.xavier_uniform is now deprecated in favor of
  I.xavier_uniform(self.fc1.weight.data)
/home/workspace/models.py:47: UserWarning: nn.init.xavier_uniform is now deprecated in favor of
  I.xavier_uniform(self.fc2.weight.data)
/home/workspace/models.py:48: UserWarning: nn.init.xavier_uniform is now deprecated in favor of
  I.xavier_uniform(self.fc3.weight.data)
```

Out[4]:
```
Net(
  (conv1): Conv2d(1, 32, kernel_size=(4, 4), stride=(1, 1))
  (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout1): Dropout(p=0.1)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout2): Dropout(p=0.2)
  (conv3): Conv2d(64, 128, kernel_size=(2, 2), stride=(1, 1))
  (bn3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
  (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout3): Dropout(p=0.3)
  (conv4): Conv2d(128, 256, kernel_size=(1, 1), stride=(1, 1))
  (bn4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
  (pool4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout4): Dropout(p=0.4)
  (fc1): Linear(in_features=43264, out_features=1000, bias=True)
  (bn5): BatchNorm1d(1000, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
  (dropout5): Dropout(p=0.5)
  (fc2): Linear(in_features=1000, out_features=1000, bias=True)
  (bn6): BatchNorm1d(1000, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
  (dropout6): Dropout(p=0.6)
  (fc3): Linear(in_features=1000, out_features=136, bias=True)
)
```

0.4 Keypoint detection

Now, we'll loop over each detected face in an image (again!) only this time, you'll transform those faces in Tensors that your CNN can accept as input images.

0.4.1 TODO: Transform each detected face into an input Tensor

You'll need to perform the following steps for each detected face: 1. Convert the face from RGB to grayscale 2. Normalize the grayscale image so that its color range falls in [0,1] instead of [0,255] 3. Rescale the detected face to be the expected square size for your CNN (224x224, suggested) 4. Reshape the numpy image into a torch image.

Hint: The sizes of faces detected by a Haar detector and the faces your network has been trained on are of different sizes. If you find that your model is generating keypoints that are too small for a given face, try adding some padding to the detected `roi` before giving it as input to your model.

You may find it useful to consult to transformation code in `data_load.py` to help you perform these processing steps.

0.4.2 TODO: Detect and display the predicted keypoints

After each face has been appropriately converted into an input Tensor for your network to see as input, you can apply your `net` to each face. The ouput should be the predicted the facial keypoints. These keypoints will need to be "un-normalized" for display, and you may find it helpful to write a helper function like `show_keypoints`. You should end up with an image like the following with facial keypoints that closely match the facial features on each individual face:

```
In [ ]:
```

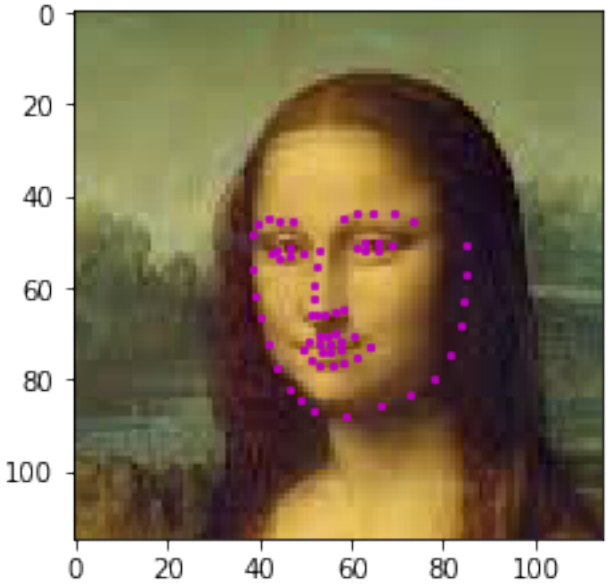
```
In [5]: import cv2
        from torch.autograd import Variable

        def show_all_keypoints(image, predicted_key_pts, gt_pts=None):
            """Show image with predicted keypoints"""
            # image is grayscale
            plt.imshow(image, cmap='gray')
            plt.scatter(predicted_key_pts[:, 0], predicted_key_pts[:, 1], s=20, marker='.', c='m
            # plot ground truth points as green pts
            if gt_pts is not None:
                plt.scatter(gt_pts[:, 0], gt_pts[:, 1], s=20, marker='.', c='g')

        image_copy = np.copy(image)

        # loop over the detected faces from your haar cascade
        for i, (x,y,w,h) in enumerate(faces):

            # Select the region of interest that is the face in the image

            roi = image_copy[y-30:y+h+30, x-30:x+w+30]
```

```
h, w = roi.shape[0], roi.shape[1]

roi_disp = np.copy(roi)

## TODO: Convert the face region from RGB to grayscale
roi = cv2.cvtColor(roi, cv2.COLOR_BGR2GRAY)
## TODO: Normalize the grayscale image so that its color range falls in [0,1] instea
roi = roi / 255.0
## TODO: Rescale the detected face to be the expected square size for your CNN
roi = cv2.resize(roi, (224, 224)).reshape(224, 224, 1)
## TODO: Reshape the numpy image shape (H x W x C) into a torch image shape (C x H x
roi = torch.from_numpy(roi.transpose(2, 0, 1))
## TODO: Make facial keypoint predictions using your loaded, trained network
## wrap each face region in a Variable and perform a forward pass to get the predict

roi_copy = np.copy(roi)

# Evaluation mode

# convert images to FloatTensors
roi = roi.type(torch.FloatTensor)
roi = roi.unsqueeze(0) # convert to 4D tensor with batch size of 1
# forward pass to get outputs
net.eval()
output_pts = net.forward(Variable(roi, volatile=True))
# reshape to batch_size x 68 x 2 pts
output_pts = output_pts.view(68, 2)
# Convert back to numpy
output_pts = output_pts.data.numpy()

print("outputs_pts.shape:", output_pts.shape)

## TODO: Display each detected face and the corresponding keypoints

print("roi_copy.shape:", roi_copy.shape)
roi = np.transpose(roi_copy, (1, 2, 0))

# undo normalization of keypoints
output_pts = (output_pts * 50.0) + 100

# undo rescaling (to display on original roi image)
output_pts = output_pts * (w / 224, h / 224)

plt.figure(figsize=(10, 8))
if (i == 0):
    plt.subplot(2, 1, 1)
    show_all_keypoints(np.squeeze(roi_disp), output_pts)
elif (i==1):
```

```
                plt.subplot(2, 1, 2)
                show_all_keypoints(np.squeeze(roi_disp), output_pts)
            plt.show()
```

```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:46: UserWarning: volatile was remov
```

```
outputs_pts.shape: (68, 2)
roi_copy.shape: (1, 224, 224)
```



In [ ]:

In [ ]:

In [ ]: