Real Face to Anime Face Image Translation with CycleGAN

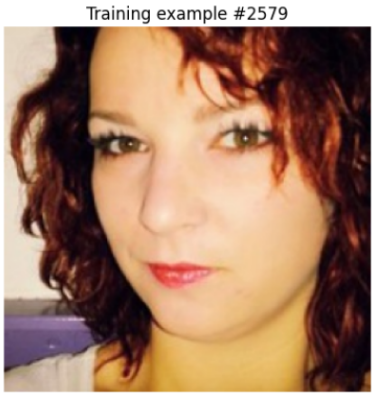
Objective:

The objective of this project is to create a CycleGAN network which is able to generate anime faces from real faces. The generated faces should not only look like anime faces, but should also bear a close resemblance to the input real face.

Methods:

Data Preparation / Loading:

I first define a FacesDataset class, which will contain the images of real and anime faces. A call to the __getitem__() method returns a tuple made up of one real face and one anime face. Before the images are returned, however, they are pre-processed. The pre-processing includes resizing the image to 128x128 and then normalizing it. I resize the image to be 128x128 in order to reduce computation time, and I normalize the input to speed up training. A facesLoader dataloader is then created using an instance of the above FacesDataset class. To make sure the above is working correctly, I visualize a couple of samples from the dataset using my helper function visualize_samples(). Before the sample images are displayed, they are un-normalized by dividing each tensor value by 2 then adding 0.5. A sample real and anime face is shown below.



Models:

My CycleGAN model consists of 2 generator models for anime and real faces, as well as 2 discriminator models for anime and real faces. The model architecture for each generator model is identical, as is the model architecture for each discriminator model.

Generator Model Architecture:

Following the guidance of the given CycleGAN paper, my generator models consist of an input layer, two downsampling layers, six residual layers (since I downsized the input image to 128x128), and three upsampling layers.

The input layer is a convolutional layer with 3 input channels (because we are dealing with RGB images), 64 output channels, a kernel size of 7, stride 1, and padding 3. This convolutional layer is followed by instance normalization, and then the ReLU activation function, as suggested by the CycleGAN paper.
Each downsampling layer is a strided convolutional layer with output channels equal to double the amount of input channels, a kernel size of 3, stride 2, and padding 1. ReLU activation functions are applied to the output of each downsampling layer.
Each residual layer consists of two strided convolutions with input channels equal to output channels, kernel size 3, and padding 1. ReLU activation functions are applied to the output of each of these convolutions. After both convolutions in the residual layer, the original input to the residual layer is then concatenated to the result.
Each upsampling layer consists of a deconvolutional layer with output channels equal to half of its input channels, a kernel size of 3, stride 2, and padding 1. This deconvolutional layer is followed by instance normalization, and then the ReLU activation function.

Discriminator Model Architecture:
Following the guidance of the given CycleGAN paper, my discriminator models consist of four convolutional layers with output channels equal to double the number of input channels (except for the first layer which has 3 input channels 64 output channels), a kernel size of 4, stride 2, and padding 1. Each convolution is followed by instance normalization and leaky ReLU with a negative slope of 0.2, as suggested by the CycleGAN paper.
Following these four convolutional layers is a final convolutional layer with only one output channel, kernel size 4, stride 1, and padding 1. After this final convolutional layer is a sigmoid activation function which maps the output to a binary label, corresponding to whether or not the discriminator model was fooled.

Saving / Loading Models:
Because training these models takes a lot of compute time, I implemented 2 helper functions, saveModel() and loadModel(), so that the models could be trained in chunks.
The saveModel() function saves a models state dictionary, the model's optimizer state dictionary, and total number of epochs that the model has currently been trained on.
The loadModel() function loads a model that was saved using saveModel().

Training:
The main training loop consists of training the anime face discriminator and the real face discriminator on a face generated by their generator counterparts. The generator models are then trained with the goal of fooling the discriminator models, while still looking like the input image. This is achieved by not only including losses from the discriminators in the generator's loss function, but also including losses from cycling. The loss from cycling is a measure of how different a face which has been generated into a face from the other domain (anime or real) and

then back again is from the original input face. Following the guidance of the CycleGAN paper, mean squared error loss is used for the discriminator losses, mean absolute error is used for the cycling losses, and cycling losses are scaled up by a factor of 10 (lambda = 10). Based on some initial tests, I decided to use a learning rate of 1e-5, much smaller than the learning rate of 0.0002 suggested by the cycleGAN paper. I save all models after each epoch.

Results:

The results of my CycleGAN implementation after 62 epochs of training are shown below. Although the generated anime faces are not yet fully convincing, they contain many anime-like elements. The results would improve in quality with more training (i.e. 100 or 200 epochs).