MAIS202 Final Project Deliverable 3: Artistic Style Transfer

Rosie Zhao

1 Final Training Results

Following deliverable 2's results after implementing a functional model, I explored hyperparameter tuning and deploying the model on the cloud. After being fortunate enough to gain access to a Microsoft Azure DSVM, I was able to generate $512px \times 512px$ images with my model, and the runtime for each image was roughly 1-2 minutes. A few of the hyperparameters I experimented with are outlined below, and were included in my slide presentation for CUCAI 2019 in Kingston, Ontario:

1.1 Hyperparameter: Model Pooling Layers

The original paper by Gatys et al. (2015) reported that they replaced the max pooling layers of the VGG-19 network with average pooling because it "improves the gradient flow and one obtains slightly more appealing results". I tried it out and received rather poor results; although it's possible that the process required more iterations, the loss function did not seem to be converging to a minimum and my results using max pooling were already decent with less iterations; hence, all images generated used the original model's max pooling layers.

1.2 Hyperparameter: Content:Style Weight: α/β

(Insert image) Recall the general loss function we are minimizing is a weighted sum because two different loss functions that track content and style; this is a parameter specified by the user, and although the original Gatys paper specified a weight of 10e-3, I ended up using a weight of 10e-6. This was a reasonable adjustment to make because the style of the paintings that Gatys used are much more texturized with less defined edges, whereas Ghibli images have softer features but clearer edges; hence I needed to emphasize minimizing style loss in comparison to the content image.

1.3 Hyperparameter: Input Images

The original implementation fed in a third image (the eventual output image) which was simply a randomly-generated white noise image, and allowed the model to generate the resulting image through a larger number of iterations; alternatively, one can use the content image itself as the input and "work towards" conveying the input image in that style. The latter implementation is what I settled on using, because the images generated from white noise were nowhere near as developed– it likely required a much larger number of steps.

1.4 Hyperparameter: Step Sizes

Step sizes is another number that is specified by the user at each iteration of image generation; I tested several numbers from the set 100, 300, 500, 1000 and settled upon 300 steps since images appeared to be developed enough at this stage (and took relatively low amount of time to generate) and grey artifacts were appearing after 500 and 1000 steps.

1.5 Overall Model

In light of these results, output images are "naively" generated by the model; in the Gatys 2015 paper, the tradeoff for the slow running time and "feature bleeding" is the versatility of such a model. Any two arbitrary images can be fed into the network and an image will be generated. Since my project specifically manipulates Studio Ghibli images, I didn't think that the working implementation I have uses the most of the specific context of my project. I looked into image segmentation, which was implemented in a follow-up paper by Gatys(2016), however I was having trouble implementing the guided gram matrix and I manually used GIMP to generate the segmention masks (before removing intermediate pixel values with K-means).

In the past week, I've been looking at FAST style transfer, which uses feed-forward networks to generate an image; although the loss functions are defined similarly, this contrasts from Gatys' original implementation because a network is trained for a specific style image and subsequently image generation is supposedly 1000x faster than Gatys' implementation with generating images on top on a pre-trained network. In a way, this method seems better fitted to the final web app I want to deploy for my model: a landing site that is specifically tasked for generating "Ghibli-style" images (given a selection of pre-trained models– more details below) that will be able to generate images much more quickly because a specific style image has been trained on. I've been looking into implementing such a model on Pytorch, and I would really like to optimize my methods for the final project. I still have my original model that can be deployed, but I would like to explore fast style transfer further so I am planning on seeing if I can get something working by this weekend and using that implementation instead. There is some overlap between the implementations like starting from a pretrained VGG network, but training time (6 hours according to online sources) is specific to fast style transfer.

2 Final Demonstration Proposal

I plan on demonstrating my model through a web application; see attached for the detailed layout. As stated above, I think fast style transfer is probably the ideal model to be deployed. I plan on having users upload an image (for the content) and they can select an image (in the fast neural style transfer implementation, also choosing a pre-trained model) as the input style with a radio button form. Upon confirmation, the image is generated and can be downloaded to the computer, so the web app will have two pages, with one directed to another through a post request. As demonstrated in the tutorial, I will be using Flask to deploy my model and I am exploring methods to possibly run the script on a server– the process is not too taxing on my computer's CPU however.

Some resources I am using:

https://medium.com/datadriveninvestor/deploy-your-pytorch-model-to-production-f69460192217

https://towardsdatascience.com/designing-a-machine-learning-model-and-deploying-it-using-flask-on-heroku-9558ce6bde7b