CSE 599 G1 Project Milestone
Mangafy: Style Transfer from manga to image

Tun Sheng Tan
netid: tunsheng
tunsheng@uw.edu

Current available neural art style transfer does not do a good job at transfering the style from a manga scene to a real image. An example of manga style transfer using DeepArt [1] is shown in Figure 1. The app is able to transfer the texture from the hair and clothe correctly but the skin and eye texture are left to be desired. CartoonGAN [2] and CycleGAN [3] shows very promising results in transfering cartoon style to images and videos due to the specialized training dataset. Inspired by that, we set out to build a style transfer neural network based on CartoonGAN using a custom database consisting of only manga style images and portrait like photos.

The styling images are scrapped from anime-pictures.net using imgbrd-grabber while the real photos are scrapped from instagram using instagram-scraper. The database consists of 5000 styling images and 5000 real photos. The training images are resized to 256x256. The raw images in JPEG are formatted into Hierarchical Data Format.

A simple baseline for this problem is the convolution neural network based style transfer using DeepArt [1] with the result shown in Figure 1. In addition, we will use the original CartoonGAN architecture as a reference for future modifications depending on the results we get from the model.

To tackle the problem, we will be using a generative-adversarial network (GAN) for style transferring. This is different from the baseline because the model used is based on convolution neural network which requires a style image and a target image. The approach is more flexible than GAN method but it does not yield satisfactory result for manga style transfer. GAN will be able to transfer manga style images without a styling image input but it is limited to the style used in training. The network design that we are using is based on CartoonGAN. An overview of CartoonGAN design is shown in Figure 2. For a more detail architecture used, refer to the Appendix.

The generator network (G) uses a convolution layer as input, then follow by two down-sampling convolution layers. In the middle, we have a residual block, follow by two up-sampling convolution layers and finally a convolution layer to generate the output. The discriminator network (D) consists of a series of convolution layer and the final layer consists of a sigmoid activation function. A special feature of CartoonGAN is the use of a VGG-16 network (VGG) for pretraining the generator and to act as a feature mapping during training.

The loss function for CartoonGAN consist of two parts: a edge-promoting adversarial loss L_{adv} and a content preserving loss L_{con}.

$$L_{adv}(G, D) = log[D(c)] + log[1 - D(e)] + log[1 - D(G(p))] \tag{1}$$

$$L_{con}(G, D) = \|VGG(G(p)) - VGG(p)\|_1 \tag{2}$$

The set of cartoon images are denoted by c and the edge-removed cartoon images are denoted by e. The target images for style transfer are denoted by p. The addition of a edge-removed image set is to help the CartoonGAN to generate images with clear edges. These edge-removed images is generated using standard Canny edge dector with a gaussian smoothing. A content loss function using L2 norm difference between the original image and generated image tends to generate

artifacts on the generated image. Instead of comparing pixel differences, a comparison between high level features such as the one generated by VGG provides a better perceptual reconstruction. In addition, the authors of the CartoonGAN shows that a sparse regularization is better at detecting features in an image [2]. In our experiment, we tried using VGG-16 but the output is slightly inferior to VGG-19 which is used by the authors. Following their choice of feature map, we use the 'conv$4'_4 layer of VGG - 19 in our training$.

Currently, we managed to pre-train the Generator using VGG to reproduce images as shown in Figure 3. However, we the reproduced images has an artifact. We are trying to debug this issue but have no luck so far. We are still developing the training and testing procedure for the CartoonGAN. The delay was due to some issues that we faced during the data preparation procedure. Getting the desired manga styled images is hard due to copyright issues and the time constraint (too time consuming to remove text from manga), we relax the requirement for manga style images to include anime styled images. During the data loading stage on Google Colab, we had trouble loading the data due to memory problem. The maximum batch size we could use is 10.

In summary, the novelty of this project is the construction of a manga portrait styled database for training a CartoonGAN specialized in style transfer for photos with subjects (person/animal).

The following is the timeline for this project:

- Week 1: Preparing database **[DONE]**
- Week 2:
  1. Data Loading **[DONE]**
  2. Define Model **[DONE]**
  3. Define Pre-Training, Training and Testing Procedure **[IN PROGRESS]**
  4. Training
- Week 3:
  1. Compare result with DeepArt
  2. Improve CartoonGAN
- Week 4: Prepare poster

References

[1] Matthias Bethge, Alex Ecker, Leon Gatys, Łukasz Kidziński, and Michał Warchoł. Deepart. URL `https://deepart.io`. [1, 8]

[2] Yang Chen, Yu-Kun Lai, and Yong-Jin Liu. Cartoongan: Generative adversarial networks for photo cartoonization. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018. URL `http://openaccess.thecvf.com/content_cvpr_2018/html/Chen_CartoonGAN_Generative_Adversarial_CVPR_2018_paper.html`. [1, 2, 9]

[3] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, 2017. [1]

Appendix: CartoonGAN architecture

Generator

```
Generator(
  (input_layer): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.2)
    (2): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (3): LeakyReLU(negative_slope=0.2)
    (4): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False,
     track_running_stats=False)
    (6): LeakyReLU(negative_slope=0.2)
    (7): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (8): LeakyReLU(negative_slope=0.2)
    (9): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
     track_running_stats=False)
    (11): LeakyReLU(negative_slope=0.2)
  )
  (res_block): Sequential(
    (0): ResidualBlock(
      (conv_layer): Sequential(
        (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
         track_running_stats=False)
        (2): ReLU()
        (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (4): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
         track_running_stats=False)
      )
    )
    (1): ResidualBlock(
      (conv_layer): Sequential(
        (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
         track_running_stats=False)
        (2): ReLU()
        (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (4): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
         track_running_stats=False)
      )
    )
    (2): ResidualBlock(
      (conv_layer): Sequential(
        (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
         track_running_stats=False)
        (2): ReLU()
        (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (4): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
         track_running_stats=False)
      )
    )
    (3): ResidualBlock(
      (conv_layer): Sequential(
        (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
```

```
        track_running_stats=False)
      (2): ReLU()
      (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
       track_running_stats=False)
    )
  )
  (4): ResidualBlock(
    (conv_layer): Sequential(
      (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
       track_running_stats=False)
      (2): ReLU()
      (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
       track_running_stats=False)
    )
  )
  (5): ResidualBlock(
    (conv_layer): Sequential(
      (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
       track_running_stats=False)
      (2): ReLU()
      (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
       track_running_stats=False)
    )
  )
  (6): ResidualBlock(
    (conv_layer): Sequential(
      (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
       track_running_stats=False)
      (2): ReLU()
      (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
       track_running_stats=False)
    )
  )
  (7): ResidualBlock(
    (conv_layer): Sequential(
      (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
       track_running_stats=False)
      (2): ReLU()
      (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
       track_running_stats=False)
    )
  )
)
(output_layer): Sequential(
  (0): ConvTranspose2d(256, 128, kernel_size=(3, 3), stride=(2, 2),
   padding=(1, 1), output_padding=(1, 1))
  (1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (2): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False,
   track_running_stats=False)
  (3): ReLU()
```

```
    (4): ConvTranspose2d(128, 64, kernel_size=(3, 3), stride=(2, 2),
     padding=(1, 1), output_padding=(1, 1))
    (5): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): InstanceNorm2d(64, eps=1e-05, momentum=0.1, affine=False,
     track_running_stats=False)
    (7): ReLU()
    (8): Conv2d(64, 3, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))
    (9): Tanh()
  ))
```

Discriminator

```
Discriminator(
  (layers): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.2)
    (2): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (3): LeakyReLU(negative_slope=0.2)
    (4): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
     track_running_stats=False)
    (6): LeakyReLU(negative_slope=0.2)
    (7): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (8): LeakyReLU(negative_slope=0.2)
    (9): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): InstanceNorm2d(512, eps=1e-05, momentum=0.1, affine=False,
     track_running_stats=False)
    (11): LeakyReLU(negative_slope=0.2)
    (12): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): InstanceNorm2d(512, eps=1e-05, momentum=0.1, affine=False,
     track_running_stats=False)
    (14): LeakyReLU(negative_slope=0.2)
    (15): Conv2d(512, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (16): Sigmoid()
  ))
```

VGG-19

```
VGG19(
  (model): VGG(
    (features): Sequential(
      (0): Conv2d(3,64,kernel_size=(3,3),stride=(1,1),padding=(1,1))
      (1): BatchNorm2d(64,eps=1e-05,momentum=0.1,affine=True,
       track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(64,64,kernel_size=(3,3),stride=(1,1),padding=(1,1))
      (4): BatchNorm2d(64,eps=1e-05,momentum=0.1,affine=True,
       track_running_stats=True)
      (5): ReLU(inplace=True)
      (6): MaxPool2d(kernel_size=2,stride=2,padding=0,dilation=1,ceil_mode=False)
      (7): Conv2d(64,128,kernel_size=(3,3),stride=(1,1),padding=(1,1))
      (8): BatchNorm2d(128,eps=1e-05,momentum=0.1,affine=True,
       track_running_stats=True)
      (9): ReLU(inplace=True)
      (10): Conv2d(128,128,kernel_size=(3,3),stride=(1,1),padding=(1,1))
      (11): BatchNorm2d(128,eps=1e-05,momentum=0.1,affine=True,
       track_running_stats=True)
      (12): ReLU(inplace=True)
      (13): MaxPool2d(kernel_size=2,stride=2,padding=0,dilation=1,ceil_mode=False)
```

```
  (14): Conv2d(128,256,kernel_size=(3,3),stride=(1,1),padding=(1,1))
  (15): BatchNorm2d(256,eps=1e-05,momentum=0.1,affine=True,
   track_running_stats=True)
  (16): ReLU(inplace=True)
  (17): Conv2d(256,256,kernel_size=(3,3),stride=(1,1),padding=(1,1))
  (18): BatchNorm2d(256,eps=1e-05,momentum=0.1,affine=True,
   track_running_stats=True)
  (19): ReLU(inplace=True)
  (20): Conv2d(256,256,kernel_size=(3,3),stride=(1,1),padding=(1,1))
  (21): BatchNorm2d(256,eps=1e-05,momentum=0.1,affine=True,
   track_running_stats=True)
  (22): ReLU(inplace=True)
  (23): Conv2d(256,256,kernel_size=(3,3),stride=(1,1),padding=(1,1))
  (24): BatchNorm2d(256,eps=1e-05,momentum=0.1,affine=True,
   track_running_stats=True)
  (25): ReLU(inplace=True)
  (26): MaxPool2d(kernel_size=2,stride=2,padding=0,dilation=1,ceil_mode=False)
  (27): Conv2d(256,512,kernel_size=(3,3),stride=(1,1),padding=(1,1))
  (28): BatchNorm2d(512,eps=1e-05,momentum=0.1,affine=True,
   track_running_stats=True)
  (29): ReLU(inplace=True)
  (30): Conv2d(512,512,kernel_size=(3,3),stride=(1,1),padding=(1,1))
  (31): BatchNorm2d(512,eps=1e-05,momentum=0.1,affine=True,
   track_running_stats=True)
  (32): ReLU(inplace=True)
  (33): Conv2d(512,512,kernel_size=(3,3),stride=(1,1),padding=(1,1))
  (34): BatchNorm2d(512,eps=1e-05,momentum=0.1,affine=True,
   track_running_stats=True)
  (35): ReLU(inplace=True)
  (36): Conv2d(512,512,kernel_size=(3,3),stride=(1,1),padding=(1,1))
  (37): BatchNorm2d(512,eps=1e-05,momentum=0.1,affine=True,
   track_running_stats=True)
  (38): ReLU(inplace=True)
  (39): MaxPool2d(kernel_size=2,stride=2,padding=0,dilation=1,ceil_mode=False)
  (40): Conv2d(512,512,kernel_size=(3,3),stride=(1,1),padding=(1,1))
  (41): BatchNorm2d(512,eps=1e-05,momentum=0.1,affine=True,
   track_running_stats=True)
  (42): ReLU(inplace=True)
  (43): Conv2d(512,512,kernel_size=(3,3),stride=(1,1),padding=(1,1))
  (44): BatchNorm2d(512,eps=1e-05,momentum=0.1,affine=True,
   track_running_stats=True)
  (45): ReLU(inplace=True)
  (46): Conv2d(512,512,kernel_size=(3,3),stride=(1,1),padding=(1,1))
  (47): BatchNorm2d(512,eps=1e-05,momentum=0.1,affine=True,
   track_running_stats=True)
  (48): ReLU(inplace=True)
  (49): Conv2d(512,512,kernel_size=(3,3),stride=(1,1),padding=(1,1))
  (50): BatchNorm2d(512,eps=1e-05,momentum=0.1,affine=True,
   track_running_stats=True)
  (51): ReLU(inplace=True)
  (52): MaxPool2d(kernel_size=2,stride=2,padding=0,dilation=1,ceil_mode=False)
)
(avgpool): AdaptiveAvgPool2d(output_size=(7,7))
(classifier): Sequential(
  (0): Linear(in_features=25088,out_features=4096,bias=True)
  (1): ReLU(inplace=True)
  (2): Dropout(p=0.5,inplace=False)
  (3): Linear(in_features=4096,out_features=4096,bias=True)
  (4): ReLU(inplace=True)
```

```
      (5): Dropout(p=0.5,inplace=False)
      (6): Linear(in_features=4096,out_features=1000,bias=True)
    )
  )
)
```

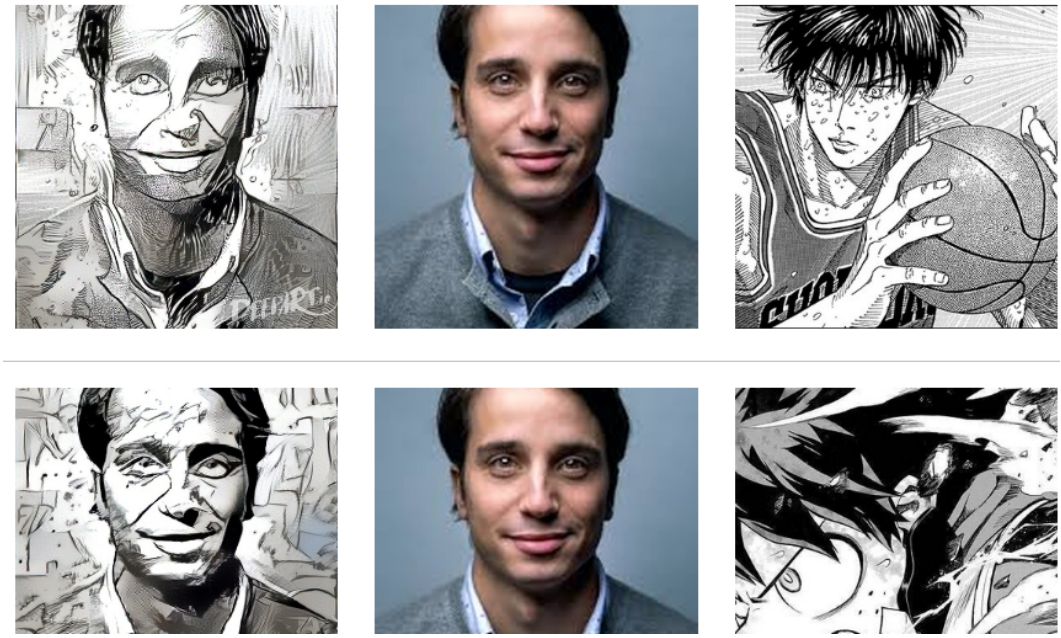

Figure 1: Example of style transfer using DeepArt with two scenes from two different mangas [1] applied to a portrait image. (Right) Input style (Middle) Input image (Left) Output

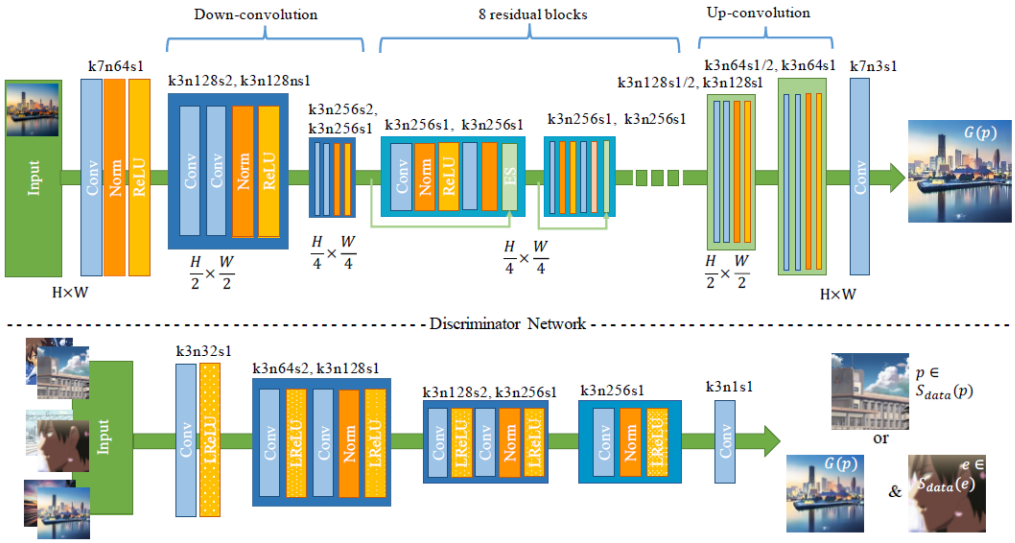

Figure 2: CartoonGAN architecture for generator (top) and discriminator (bottom) networks [2]. The parameters for the convolution layer (Conv) are labeled by kernel size k, output channel n and stride s. Batch normalization layer is labeled *Norm* and elementwise sum is labled *ES*.

Figure 3: Reconstruction of real image (left) using generator pretrained with VGG-19 after 400 epochs. The reconstructed image (right) is good but has an artifact.

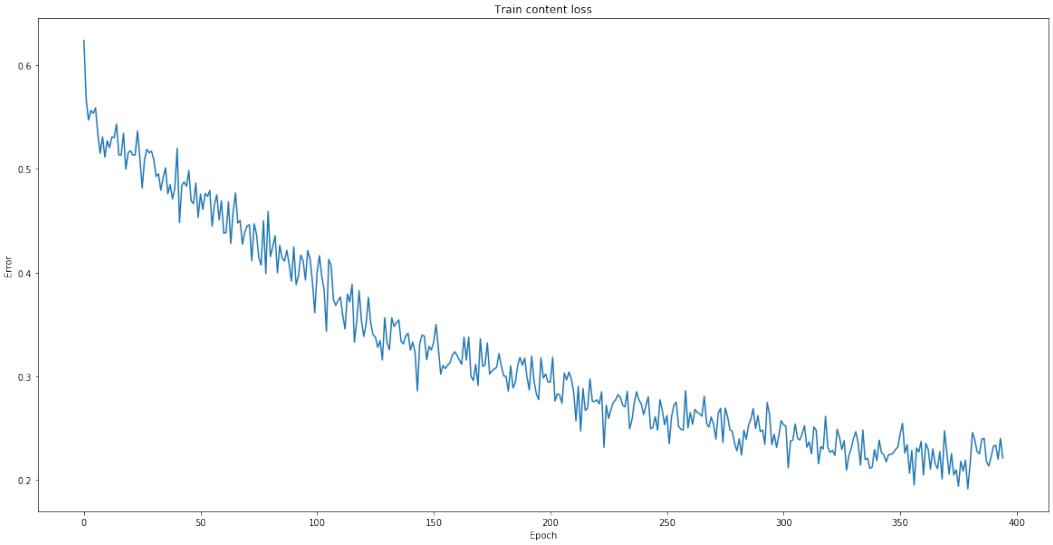

Figure 4: Pretraining reconstruction losses for the generator using VGG-19 with 400 epochs.