STYLE TRANSFER WITH ADAPTIVE INSTANCE NORMALIZATION

b09902008 葉乃瑄 b09902131 黃紹軒

1. INTRODUCTION & BACKGROUND

1.1. Introduction

When deciding the topic for our final project, we browsed through some former students' projects. Among them, we found many topics about style transfer using neural network technique, but most of their transfer models either took large amount of time to train or could only do one-to-one style transfer. With further study, we discovered a recent research about style transfer architecture using Adaptive Instance Normalization, which achieves both short training time and arbitrary style transfer. During our project, we first implemented the model architecture in the paper using Adaptive Instance Normalization and pre-trained VGG encoder. Afterwards, we managed to use DenseNet as our encoder. In our report, we will first introduce the evolution of normalization method used in style transfer. Then, we will show our experiment result and compare our method using DenseNet with the one in the paper.

1.2. Background

1.2.1. Batch Normalization

The seminal work of Ioffe and Szegedy introduced a batch normalization (BN) layer that significantly ease the training of feed-forward networks by normalizing feature statistics. BN layers are originally designed to accelerate training of discriminative networks, but have also been found effective in generative image modeling. Given an input batch $x \in R^{N\times C\times H\times W}$, BN normalizes the mean and standard deviation for each individual feature channel:

$$BN(x) = \gamma(\frac{x-\mu(x)}{\sigma(x)}) + \beta$$

where $\gamma, \beta \in R^C$ are affine parameters learned from data; $\mu(x), \sigma(x) \in R^C$ are the mean and standard deviation, computed across batch size and spatial dimensions independently for each feature channel:

$$\mu_n c(x) = \frac{1}{NHW}\sum_{n=1}^{N}\sum_{h=1}^{H}\sum_{w=1}^{W} x_{nchw}$$

$$\sigma_c(x) = \sqrt{\frac{1}{NHW}\sum_{n=1}^{N}\sum_{h=1}^{H}\sum_{w=1}^{W}(x_{nchw}-\mu_c(x))^2 + \epsilon}$$

BN uses mini-batch statistics during training and replace them with popular statistics during inference, introducing discrepancy between training and inference. Batch renormalization was recently proposed to address this issue by gradually using popular statistics during training. As another interesting application of BN, Li *et al.* found that BN can alleviate domain shifts by recomputing popular statistics in the target domain. Recently, several alternative normalization schemes have been proposed to extend BN's effectiveness to recurrent architectures.

1.3. Instance Normalization

In the original feed-forward stylization method, the style transfer network contains a BN layer after each convolutional layer. Surprisingly, Ulyanov *et al.* found that significant improvement could be achieved simply by replacing BN layers with IN layers:

$$IN(x) = \gamma(\frac{x-\mu(x)}{\sigma(x)}) + \beta$$

Different from BN layers, here $\mu(x)$ and $\sigma(x)$ are computed across spatial dimensions independently for each channel and each sample:

$$\mu_c(x) = \frac{1}{HW}\sum_{h=1}^{H}\sum_{w=1}^{W} x_{nchw}$$

$$\sigma_c(x) = \sqrt{\frac{1}{HW}\sum_{h=1}^{H}\sum_{w=1}^{W}(x_{nchw}-\mu_{nc}(x))^2 + \epsilon}$$

Another difference is that IN layers are applied at test time unchanged, whereas BN layers usually replace mini-batch statistics with population statistics.

1.4. Adaptive Instance Normalization

Xun Huang and Serge Belongie propose a simple extension to IN, which is called adaptive instance normalization (AdaIN). AdaIN receives a content input x and a

style input y, and simply aligns the channel-wise mean and variance of x to match those of y. Unlike BN or IN, AdaIN has no learnable affine parameters. Instead, it adaptively computes the affine parameters from the style input:

$$\text{AdaIN}(x, y) = \sigma(y)(\frac{x - \mu(x)}{\sigma(x)}) + \mu(y)$$

in which they simply scale the normalized content input with $\sigma(y)$, and shift it with $\mu(y)$. Similar to IN, these statistics are computed across spatial locations. Intuitively, consider a feature channel that detects brushstrokes of a certain style. A style image with this kind of strokes will produce a high average activation for this feature. The output produced by AdaIN will have the same high average activation for this feature, while preserving the spatial structure of the content image. The brushstroke feature can be inverted to the image space with a feed-forward decoder. The variance of this feature channel can encoder more subtle style information, which is also transferred to the AdaIN output and the final output image. In short, AdaIN performs style transfer in the feature space by transferring feature statistics, specifically the channel-wise mean and variance. While the style swap operation is very time-consuming and memory-consuming, AdaIN layer is as simple as an IN layer, adding almost no computational cost.

2. ADAIN NETWORKS

2.1. Architecture

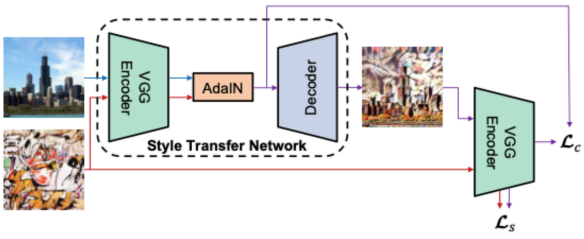

Fig. 2.1. Main model

Their style transfer network T takes a content image c and an arbitrary style image s as inputs, and synthesizes an output image that recombines the content of the former and the style latter. They adopt a simple encoder-decoder architecture, in which the encoder f is fixed to the first few layers (up to relu4 1) of a pre-trained VGG-19. After encoding the content and style images in feature space, we implement AdaIN as mentioned before to merge two images:

$$t = \text{AdaIN}(f(c), f(s))$$

A randomly initialized decoder g is trained to map t back to the image space, generating the stylized image The decoder mostly mirrors the encoder, with all pooling layers replaced by nearest up-sampling to reduce checkerboard effects.

2.2. Loss function

They use the pre-trained VGG19 to compute the loss function to train the decoder:

$$L = L_c + L_s$$

which is a weighted combination of the content loss Lc and the style loss Ls with the style loss weight . The content loss is the Euclidean distance between the target features and the features of the output image. They use the AdaIN output t as the content target, instead of the commonly used feature responses of the content image.

$$L_c = ||f(g(t))-t||_2$$

Since their AdaIN layer only transfers the mean and standard deviation of the style features, their style loss only matches these statistics.

$$L_s = \sum_{i=1}^{L} ||\mu(\phi_i(g(t)))-\mu(\phi_i(s))||_2 + \sum_{i=1}^{L} ||\sigma(\phi_i(g(t)))-\sigma(\phi_i(s))||_2$$

where each ϕ_i denotes a layer in VGG-19 used to compute the style loss. They use relu1_1, relu2_1, relu3_1, relu4_1 layers with equal weights.

3. EXPERIMENTS

3.1. Why go dense?

Because this paper was published almost 5 years ago, we noticed that the encoder they used was pre-trained VGG-19 published 8 years ago, which is a bit outdated. Thus, we did some experiments to merge AdaIN architecture with DenseNet, a CNN model famous for its ability to avoid vanishing gradient and significantly reduce number of parameters. During our experiments, We used merely 1000 content images and 10 style images to test our methods.

3.2. DenseNet Encoder-Decoder

At first, we built an encoder composed of two 6-layer dense blocks and a decoder which mirrored with encoder and replaced dense layers with convolution layers. However, we still had to retain some parts of original structure containing pre-trained VGG-19 since we needed a pre-trained model to compute proper loss and

update our model. New pipeline would be Fig. 4.1 and outcome can be seen in Fig. 4.2.

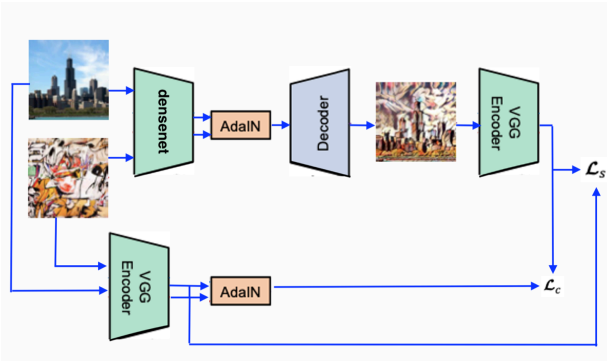

Fig. 3.1. architecture 1

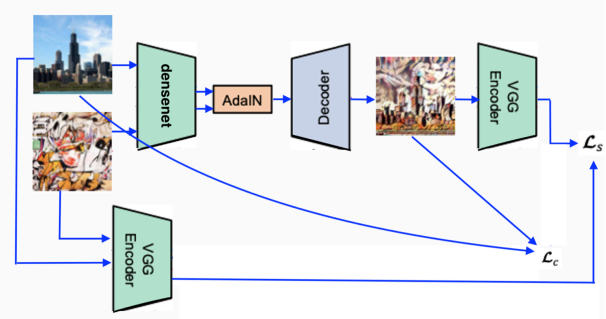

Fig. 3.3. architecture 2

Fig. 3.2. model output 1(the images from first row to last row are content images, style images, style-transfer images, model output images without AdaIN layer, in order.)

Fig. 3.4. model output 2

From Fig. 3.2, our model failed to transfer the content images. The main reason is perhaps that we reused too many parts of the original structure and used AdaIN layer once again just to calculate content loss.

3.3. DenseNet AutoEncoder

Looking into the outcome of the original model, we discovered that if we skipped AdaIN layer and used only encoder and decoder, the output picture would be very similar to content image, which implied that our encoder-decoder can be viewed as an autoencoder. Thus, we can simply calculated the mean square error between content image and output image as our new content loss (Fig. 4.3) to update the model. By doing so, we could remove most of the redundant structure.

The outcome (Fig. 4.4) this time is way better than the previous one. However, it is still far from perfect. The major problem of this method is that the mean square error we used when calculating content loss mainly ranged from 0 to 0.01, which was totally different from the one of the original model. Therefore, we had to rearrange the ratio between content loss and style loss and found a proper learning rate to scale the total loss, both of which could be tricky and might need lots of time to train. In our attempts, we found that once we gave content loss more weights, our model would do better on reconstructing content image, but could not transfer style well(Fig. 4.5). On the other hand, giving too much weights on style loss would lose the ability to encode a picture well, causing style transfer to fail (Fig. 4.6).

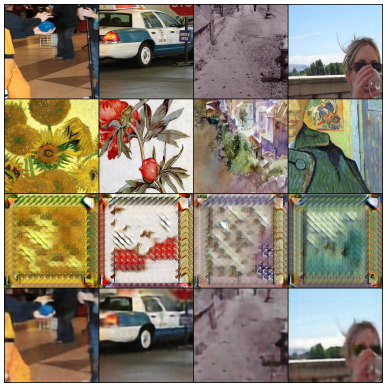

Fig. 3.5. model output weights more on content loss

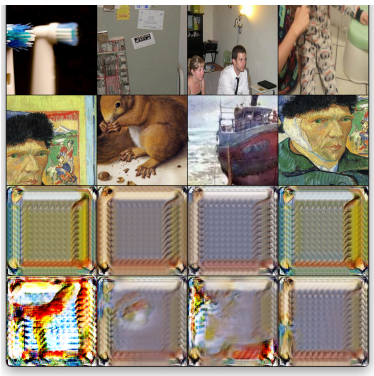

Fig. 3.6. model output weights more on style loss

3.4. DenseNet AdaIN

After our attempts, we thought that the difference between DenseNet and VGG-19 might be so huge that evaluating our dense model through a pre-trained VGG-19 was of no help to make our model better. Thus, we decided to import pre-trained DenseNet-121 of which we only used the top three dense blocks(6-12-24-layer) as our encoder and rebuilt a decoder to mirror it (Fig. 4.7). Furthermore, we used transition layers between each dense blocks to compute our style loss. However, as Fig. 4.8, the result became worse.

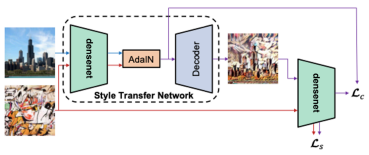

Fig. 3.7. architecture 3

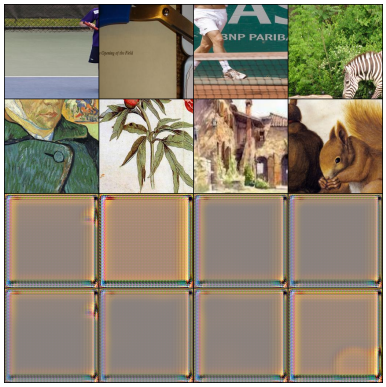

Fig. 3.8. model output 3

3.5. Simpler Decoder

After several tests and different configurations of learning rate, lambda, batch size, we realized that our decoder mirroring DenseNet encoder might be too complex. Originally, a dense layer would contain two convolution layers to compress input, then concatenated input with output and pass the result to the next layer. We realized that simply copying this structure and reversing it as our decoder layer was not a good idea. Thus, we tried to use only one convolution layer to mimic one dense layer. We then had a decoder with half of the encoder layers. After few epochs of training, we successfully merged DenseNet with AdaIN architecture using Fig. 4.1.

4. RESULTS

4.1. Training

We trained DenseNet and VGG net using MSCOCO as content images (roughly 100,000 pictures) and a variety of style images (10) from WikiArt. We used the Adam optimizer and a batch size of 8 content-style image pairs. During training, we first resized the smallest dimension of both images to 512 while preserving the aspect ratio, then randomly cropped regions of size 256 $\times$ 256. Finally, we paired each content image to a style image evenly.

4.2. Comparison of two models

4.2.1. Observation

From Fig. 4.1, we can see the difference is quiet distinct. Take the stared cat image for example, the VGG version fully transferred the original image to sketch-style image. Our DenseNet version, however, still contained some parts from the content image, like the cat's eyes are not transferred completely. For short, VGG

Fig. 4.1. two model outputs, where content images is out of training dataset

version model has better overall transfer performance, while our DenseNet version model will leave some parts untransformed.

4.2.2. Analyzation

During our training, we found that our content image loss is always lower than the one of VGG encoder model, while style image loss is always higher, which explains our observation of the final result. With further study into DenseNet, we think the defect of our transfer is resulted from DenseNet's mechanism, which concatenates every layer with all of its former layer. The mechanism will cause Dense Connectivity, by which we mean the feature will be partially inherited during every layer of DenseNet. Due to Dense Connectivity, the transferred image will keep some of the feature from our input (content image), and cause the defect of our result.

4.3. Contribution

During our project, we modified the model architecture by replacing VGG encoder with DenseNet. Furthermore, we succeeded to transfer out content image using DenseNet. Although our result does not surpass the one in the paper, we discover the traits of DenseNet and its deficiency when used in style transfer.

5. REFERENCES

Xun Huang and Serge Belongie. Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization. arXiv:1703.06868v2, 2017

6. CODE

https://github.com/yahcreepers/ICG_Final