Fast style transfer project technical report

Chenyu Wang

April 8, 2022

Contents

1 Introduction **2**

2 Dataset **2**

3 Model architecture **2**
3.1 Image transformation network 3
3.2 Loss network 3
3.3 Inference 3

4 Cloud architecture **4**
4.1 Lambda Function 4
4.2 AWS S3 Bucket 4
4.3 API configuration 4

5 Application Evaluation **5**
5.1 Work examples 5
5.2 Bad case 5

6 Conclusion **6**

1 Introduction

This project is a cloud implementation of fast-neural-style in PyTorch. Style Transfer learns the aesthetic style of a style image, usually an artwork, and applies it to another content image. This implementation mainly follows the paper [1] along with some engineer modifications in the code part for better performance. To fully present this project, we design AWS(Amazon Web Services) cloud architecture for the model inference part and leverage the front-end framework React to stylize and visualize the user-defined input image. Finally, we deploy our project to the Heroku website.

2 Dataset

The MS COCO (Microsoft Common Objects in Context) dataset[2] is a large-scale object detection, segmentation, key-point detection, and captioning dataset. The dataset consists of 328K images. In this application, COCO 2014 Training images dataset is used for pre-trained model[1].

3 Model architecture

Image style transfer problem can be modeled as image transformation tasks where a system receives a degraded image and output the desired image. For the image transformation problem, training a feed-forward convolutional neural network in a supervised manner is way more efficient than traditional algorithms. In this application, as shown in Figure 1, we train feedforward transformation networks for style transformation tasks, but here we use perceptual loss functions that depend on high-level features from a pre-trained loss network. During training, perceptual losses measure image similarities and at test-time, the transformation networks run in real-time.

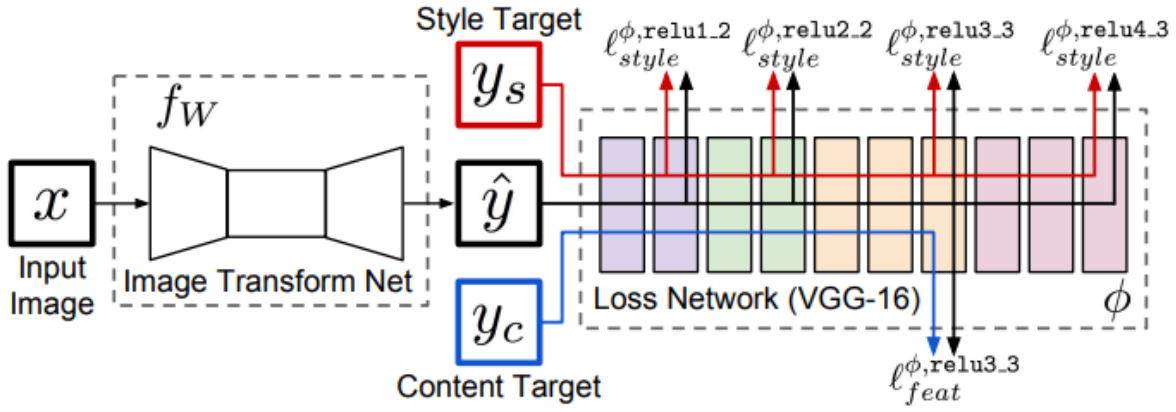

Figure 1: Model architecture

[1]See `https://cocodataset.org/#download`

3.1 Image transformation network

We train an image transformation network to transform input images into output images. Here, the image transformation network is trained using stochastic gradient descent to minimize a weighted combination of loss functions[2]:

$$W^* = \arg\min_W \mathbf{E}_{x,\{y_i\}} \left[\sum_{i=1} \lambda_i \ell_i \left(f_W(x), y_i \right) \right] \tag{1}$$

3.2 Loss network

We use a loss network pre-trained for image classification to define perceptual loss functions that measure perceptual differences in content and style between images. The loss network remains fixed during the training process. Here, we use the 16-layer VGG network [3] pre-trained on COCO 2014.

3.3 Inference

Given the style and content targets y_s and y_c and layers j and J at which to perform feature and style reconstruction, an image $\hat{y}$ is generated by solving the problem

$$\hat{y} = \arg\min_y \lambda_c \ell^{\phi,j}_{feat} \left(y, y_c \right) + \lambda_s \ell^{\phi,J}_{\text{style}} \left(y, y_s \right) + \lambda_{TV} \ell_{TV}(y) \tag{2}$$

where λ_c, λ_s, and λ_{TV} are scalars, y is initialized with white noise, and optimization is performed using L-BFGS[4]. The architecture of transform network is shown in Table 1.

Layer	Activation size
Input	$3 \times 256 \times 256$
$32 \times 9 \times 9$ conv, stride 1	$32 \times 256 \times 256$
$64 \times 3 \times 3$ conv, stride 2	$64 \times 128 \times 128$
$128 \times 3 \times 3$ conv, stride 2	$128 \times 64 \times 64$
Residual block, 128 filters	$128 \times 64 \times 64$
Residual block, 128 filters	$128 \times 64 \times 64$
Residual block, 128 filters	$128 \times 64 \times 64$
Residual block, 128 filters	$128 \times 64 \times 64$
Residual block, 128 filters	$128 \times 64 \times 64$
$64 \times 3 \times 3$ conv, stride 1/2	$64 \times 128 \times 128$
$32 \times 3 \times 3$ conv, stride 1/2	$32 \times 256 \times 256$
$3 \times 9 \times 9$ conv, stride 1	$3 \times 256 \times 256$

Table 1: Network architecture used for style transfer networks.

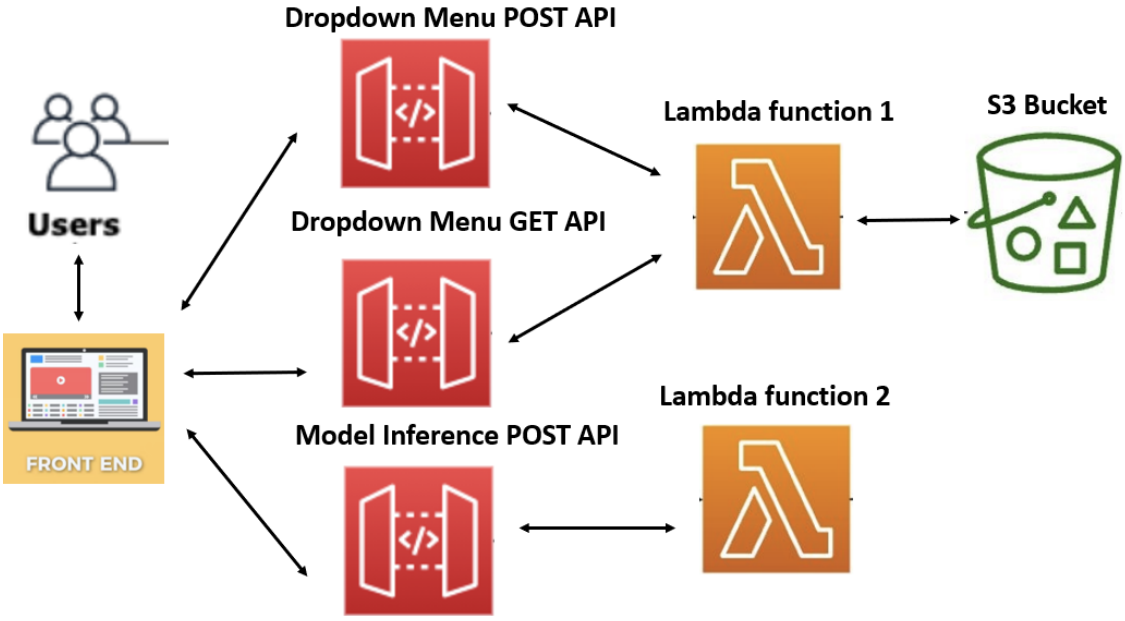

Figure 2: Model architecture

4 Cloud architecture

4.1 Lambda Function

Lambda Function 1 was created to handle the dropdown menu component for demo purposes and it was connected to two API methods.

Lambda Function 2 was built for model inference in a cloud way, it was consists of

- Lambda Layers: numpy, onnxruntime, PIL
- Zip folders: transformer.onnx, lambda_inference_function

4.2 AWS S3 Bucket

In our application, since we use lambda layers, we do not need to use S3 buckets to store library dependencies for model inference in the cloud. Instead, we create a S3 bucket to store working images for demo purposes.

4.3 API configuration

To forward incoming requests from the front end to the above Lambda functions, we design two API Gateways where the dropdown menu API Gateway includes two methods

- GET method to fetch image files list in the S3 bucket.
- POST method to download selected image for model inference.

[2]See paper [1] for detailed information

Model inference API gateway only has one Post method which is used to transform input image to target.

5 Application Evaluation

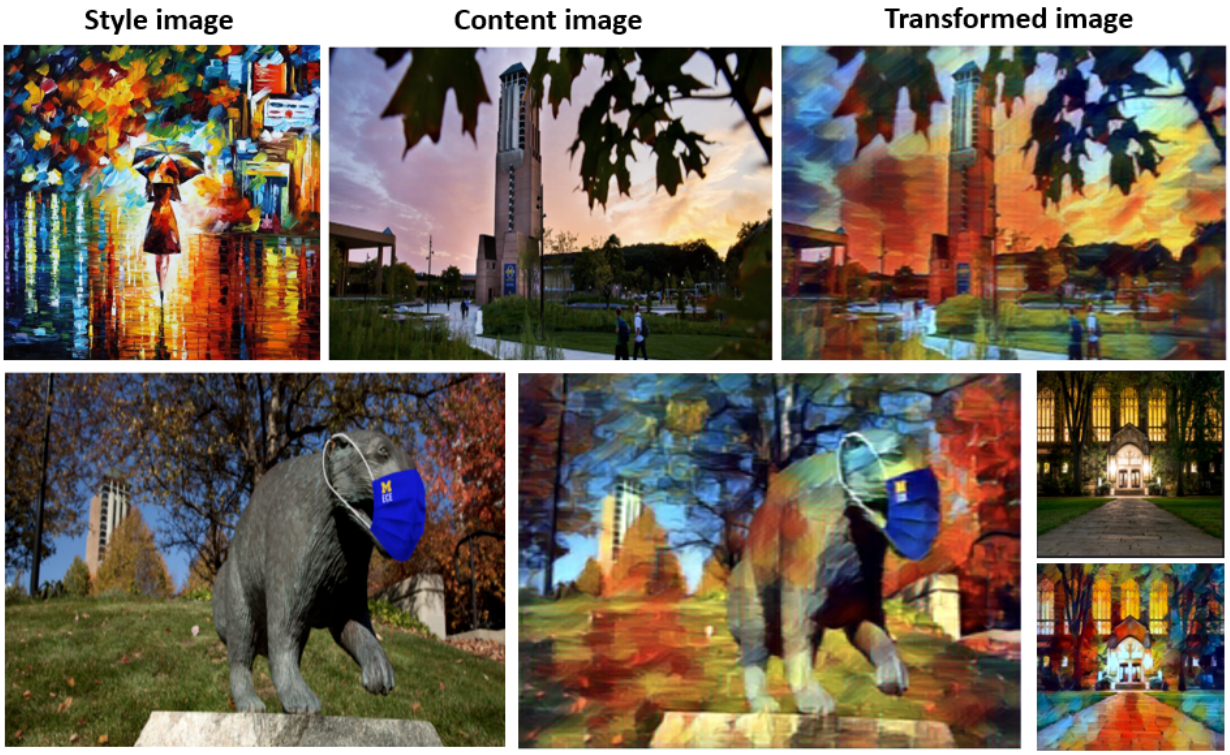

Figure 3: Examples that work well

5.1 Work examples

In our application, we choose style image *rain princess* shown in upper left in Figure3 to train transformer network. To show the model performance, we present three landscape images of UM as examples in Figure3.

5.2 Bad case

As shown in Figure 4, the example image doesn't work well. Transform network only stylized the center of the image. By our analysis, we believe it can be fixed by fine-tuning the hyperparameter λ_s shown in Eq. 2. Intuitively, we can increase λ_s to make the model-generated image close to style one but in practice, we probably need to make more engineering adjustments for those kinds of input images.

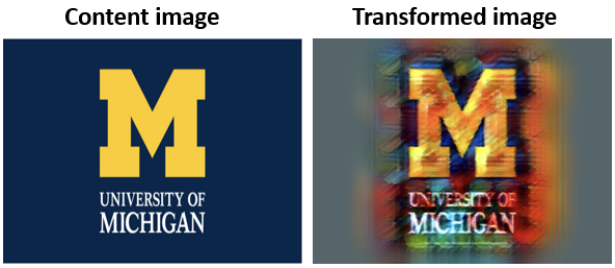

Figure 4: Example that does not work well

6 Conclusion

In this technical report, we show a React app deployed in Heroku that can do image style transformation via designed AWS cloud architecture and several APIs. The application can be found at `https://fast-transfer-style-ap.herokuapp.com/`. In the future, we will aim to improve the front-end design to make it smoother for users and also make the whole pipeline robust for long-term running.

References

[1] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In *European conference on computer vision*, pages 694–711. Springer, 2016.

[2] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[3] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[4] Ciyou Zhu, Richard H Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on mathematical software (TOMS)*, 23(4):550–560, 1997.