Neural Style Transfer: Implementation, Dash-app and more

Omkar Mehta (omehta2), Anurag Anand (anuraga2)

April 2022

1 Executive Summary

1.1 Problem Statement

Humans have mastered the art of creating unique visual experiences by harnessing the complex interplay between the content and style of an image. This is evident once we look at some of the masterpieces of the fine arts like Starry Night and The Scream. The algorithmic basis of this process existed but was limited to non-parametric methods. Then came a paper by Gatys et al. [1] that used an artificial system based on Deep Neural Networks to create artistic images. The system utilized neural representations to separate and recombine the content and the style of arbitrary images, providing a neural algorithm for the creation of artistic images.

In our project, we took inspiration from the Neural Style Transfer paper [1] and implemented a multimodal, scalable solution to the problem of style transfer. We applied a slight twist to the algorithm discussed in the paper and packaged our solution as a web application where the users can upload an image and style it based on a fixed set of styling images. This report details the problem statement, discusses the line of attack, and lists some of the interesting/surprising findings, results we landed upon during our work.

1.2 Line of Attack

The style transfer methodology mentioned in the paper [1] updated pixel values of the output image with each gradient update. This meant that for each combination of Content and Style image, the gradient had to be updated to generate the output pixel values that has the stylized image features. We illustrate this methodology, where we take a content image and apply starry nights style to it. The three pictures (content, style and output) are displayed in the panel below 1.

(a) Content Image

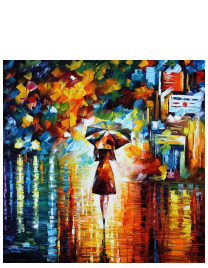
(b) Style Image

(c) Stylized Image

Figure 1: Content Image, Style Image and Stylized Image using Gatys et al.'s pixel update approach on content image

We found an issue with this implementation for our web application. Since pixel values of the content image were updated for each version, this process of stylizing was not scalable. We were required to train each time we have a new content image, making the inference on test images extremely slow. Our requirement was extremely fast inference models that would be exposed to the user's content image input and stylize it in a few seconds. So, we proposed two models, in which parameters will be trained to encode the representation of the stylized images. The trained parameters of the models are used at the inference time for texture transfer, while preserving the structure of the content image.

The two models, **VGG16** and **AONET** (Custom Model, the name that's in the code) were trained on two datasets, namely **tinyimagenet** [2] and **Validation data of COCO dataset** [3] such that the output of the model itself is a stylized image. At the end of the training process we had 20 model files (1 for each style image from 2 different architectures trained on 2 datasets). These model files encoded the representation of the style images. Designing the inference in this way worked well for us, since we were able to save the model parameters, which were then used in the dash web application for fast inference. For illustration, we show a cat image stylized with different styles using the **AONET** architecture trained on COCO validation dataset. We present one example of each style in Figure 2. These stylized images are the part of the training process in which the AONET model's output is the stylized image.



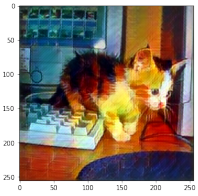
(a) Rain Princess

(b) The Scream

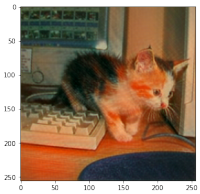
(c) The Shipwreck

(d) Udnie

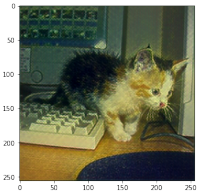
(e) Wave



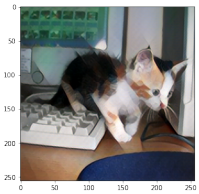
(f) Cat Stylized with Rain Princess



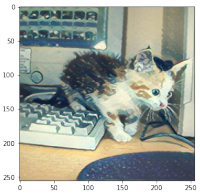
(g) Cat stylized with Scream



(h) Cat stylized with the Shipwreck



(i) Cat stylized with Udnie



(j) Cat stylized with Wave

Figure 2: Cat stylized with AONET Model trained with different styles on COCO validation

1.3 Findings

The findings that we would discuss in this section guided our approach. We observed that Gatys et al.'s approach was very slow, discussed in Section 1.2. Even though we had planned to train the models only on TinyImageNet dataset, we found out that the models needed fairly high quality images in order to get an acceptable stylized image. So, we also decided to train the models on COCO's validation dataset.

The depth of the models also have an effect on the quality of the stylized images. Even if we increase the depth of the models, we need to ensure that we provide enough memory for the training. This is discussed in Section 3.4. We also found out that the preservation of the content and style images in the output images depend highly on the content and style weights. If content weight is increased, we get better content representations in the output image, and same story goes for the style weights. This is discussed in great detail in Section 4.1.1.

We also studied the effect of different normalization methods known to us. Even though batch normalization reduces the internal covariance shift, it normalizes a batch of samples to be centered around a

single style. This is undesirable if we wanted to transfer all images to the same style, as is the case with model-based training approach. We moved our attention to Instance Normalization [4]. Instance Normalization layer is invariant to the contrast of the content image. Instance normalization can normalize the style of each individual sample to the target style. So, we would have to only worry about the content manipulation by the network.

2 Introduction

The problem of style transfer from one image to another is primarily a problem of texture transfer. The eventual goal of texture transfer is to constrain the texture synthesis from a styling image in such a way so as to preserve the semantic content of the target image. In the past, this problem has been dealt by harnessing a large range of powerful non-parametric methods. Some of these algorithms (like Efros and Freeman [5]) introduce a correspondence map that includes the features of the target such as image intensity to constrain the texture synthesis procedure. They call this process *image quilting*. In this procedure, first, they use quilting as a fast and simple texture synthesis algorithm.Second, they extend the algorithm to perform texture transfer - rendering an object with texture taken from a different object. Other methods (Hertzmann et al. [6]) used image analogies to transfer the texture from an already stylized image onto a target image. The "image analogy" framework involves two stages: a *design phase*, in which a pair of images, with one image, that appears to be the filtered version of the other, is presented as "training data" and an *application phase*, in which the learned filter is applied to some new target image in order to create an "analogous" filtered results.

While these algorithms achieve remarkable results, they have a common fundamental drawback. They used only low-level image features of the target image to inform the texture transfer. Ideally, a good style transfer algorithm should be able to extract the semantic content of the image (e.g. the objects and the general scenery) from the target image and then inform a texture transfer procedure to render the semantic content of the target by using the styling image.

Separating the content from the style in a natural image is still an extremely difficult problem. That being said, the recent advances in deep learning systems have produced powerful computer vision algorithms that learn to extract high-level semantic information from natural images. It has been shown through competitions like ILSVRC that convolutional neural networks with sufficient labelled data on specific tasks such as "Object Detection" learn to extract high-level image content in the form of generic feature representations that generalize very well across datasets.

In the neural style transfer paper [1], Gatys et al showed that the generic feature representations produced by the pre-trained convolutional neural networks can be used to independently process and manipulate the content and the style of natural images. They frame the problem of style transfer as an optimization problem based on the deep image representations. In their method, new images are formed by performing a pre-image search to match feature representations of the example image. This algorithm works well but lacks in scalability because of high inference time.

In our project, we take inspiration from the paper by Gatys et al [1], and apply a small twist to it to speed up the inference process. In their texture transfer algorithm, Gatys et al, propagated the loss to image pixel values to change the image content with each iteration. We, on the other hand, propagate the loss to the model parameters to learn a robust representation of the style image and used those representation for the texture transfer problem.

3 Details of the approach

We will compare two approaches, one with the optimization strategy of Gatys et al. [1], where the pixel values of the image are updated with respect to the loss, and the model parameters are fixed. The other approach is with model-based training, where we update the parameters of the model to provide us the stylized image. In the latter approach, we train two architectures, namely VGG16 (as encoder)

with upsampling layers and AONET (custom architecture drawing design inspiration from GoogleNet and VGG16) (downsampling, resnet, upsampling).

3.1 Gatys et al.'s Approach

We followed the exact same procedure established in the Gatys et al.'s research paper[1] for getting style transfer. The algorithm for the research paper's approach is outlined in Algorithm 1. We used pre-trained 16-layer VGG network[7] for the feature extraction, instead of 19-layer VGG network, for computational constraints.

The content image is passed through VGG16 model to get the content representation. A layer with N_l filters has N_l feature maps each of size M_l, where M_l is the height times the width of the feature map[1]. So, the responses in a layer l can be stored in a matrix $P_l \in \mathbb{R}^{N_l x M_l}$. This is what we mean by the content representation or any representation that passes through VGG16 model. We chose a desired content layer c (usually higher layer) to get us the P_c from any of the P_l content representations, because higher layers capture the high-level content in terms of objects.

For style representations, we chose five different style layers of the VGG16 model. For each style layer, we got style representation of the style image. We made a list S of style representations of five chosen style layers. We also computed the gram matrix of the style representations in Equation 1. Gram matrix A^l represents feature correlations, where A^l_{ij} is the inner product between the vectorized feature maps i and j in layer l.

$$A^l_{ij} = \frac{(\sum_k S^l_{ik} S^l_{jk})}{(N_l x M_l x C)} \tag{1}$$

where C is channel.

These feature correlations capture the texture information but not the global arrangement. Now, we store the gram matrix for all five chosen style layers in a list A.

For style transfer of style image on content image, we generated a new image that matched the content representations of the content image and the style representation of style image, by first initializing the generated image to be the content image itself. Similarly, we passed it through the VGG16 model to get F as the features of the generated image. F_c represents the feature of the chosen content layer. F_s represents the list of features from chosen style layers. We computed G as list of gram matrix of all the features in F_s.

We then minimised total loss, which is the sum of the content loss and style loss. The content loss is defined as the mean squared-error loss between the two feature representations of the content image and the generated image, computed in Equation 2.

$$\mathcal{L}_{content} = w_c * mse\_loss(P_c, F_c) \tag{2}$$

where w_c is the content weight.

The style loss is also defined as the mean squared-error loss between the gram matrices of the two style representations of the style image and the generated image, computed in Equation 3.

$$\mathcal{L}_{style} = w_s * \sum_{l}^{style\_layers} mse\_loss(A_l, G_l) \tag{3}$$

where A_l, G_l represent the gram matrix of the style representations of the style image and generated image for the layer l, and w_s is the style weight.

The loss that we minimized was the total loss $\mathcal{L}_{total}$. The gradient with respect to the pixel values of the generated image $\frac{\partial \mathcal{L}_{total}}{\partial x}$, where x is the generated image, was used as input for numerical optimization strategy. The visual representation of what we did is succinctly shown in Figure 3. The stylized images at various epochs are shown in Figure 4.

Algorithm 1 Style Transfer with Image as trainable variable

Require:

Pretrained VGG16 model: $vgg\_fe()$ ▷ Feature Extractor
style_image, content_image, epochs, learning_rate, optimizer()
mse_loss() ▷ Mean Squared Error as Loss
gram_matrix() ▷ Gram Matrix Function to compute feature correlations
style_layers ▷ A list of style layers, usually of size 4
content_layer ▷ Desired Content Layer
w_c , w_s ▷ Content Weight, Style Weight, resp.
procedure STYLE_TRANSFER(vgg_fe, style_image, content_image, epochs, learning_rate, optimizer, mse_loss, style_layers, content_layer)
    $P_c = vgg\_fe(content\_image)[content\_layer]$ ▷ Get Content Features
    $S = [vgg\_fe(style\_image)[l]$ for l *in* $style\_layers]$ ▷ Get a list of Style Features of chosen style layers
    $A = [gram\_matrix(S[i])\ for\ i\ in\ len(S)]$ ▷ Get list of target Gram Matrix of chosen style layers
    $generated\_image = content\_image.copy()$ ▷ Trainable parameter
    $optim = optimizer([generated\_image], lr = learning\_rate)$ ▷ Initialize the optimizer
    $vgg\_fe(style\_image)$ ▷ Get Style Features
    for each epoch **do**
        $F = vgg\_fe(generated\_image)$ ▷ Get features of generated image
        $F_c = F[content\_layer]$ ▷ Get desired feature from desired content layer
        $F_s = [F[l]\ for\ l\ in\ style\_layers]$ ▷ list of desired features of generated image from style layers
        $L_{content} = mse\_loss(P_c, F_c) * w_c$ ▷ Get Content loss, multiplied by Content Weight
        $G = [gram\_matrix(F_s[i])\ for\ i\ in\ len(F_s)]$ ▷ Get list of Generated Image's Gram Matrix
        $L_{style} = 0$ ▷ Initialize style loss to 0
        for each i in len(A) **do**
            $L_{style} += mse\_loss(A[i], G[i]) * w_s$ ▷ Multiply by Style Weight
        end for
        $L_{total} = L_{content} + L_{style}$ ▷ Total loss
        update the pixels of generated_image wrt L_{total}
    end for
    return *generated_image*
end procedure

3.2 VGG16 and AONET Approach

We trained two models: pretrained VGG16 with upsampling layers (model_vgg16) and AONET with upsampling layers on two datasets (model_aonet). The architectures of these models are shown in Figure 5a and 5b.

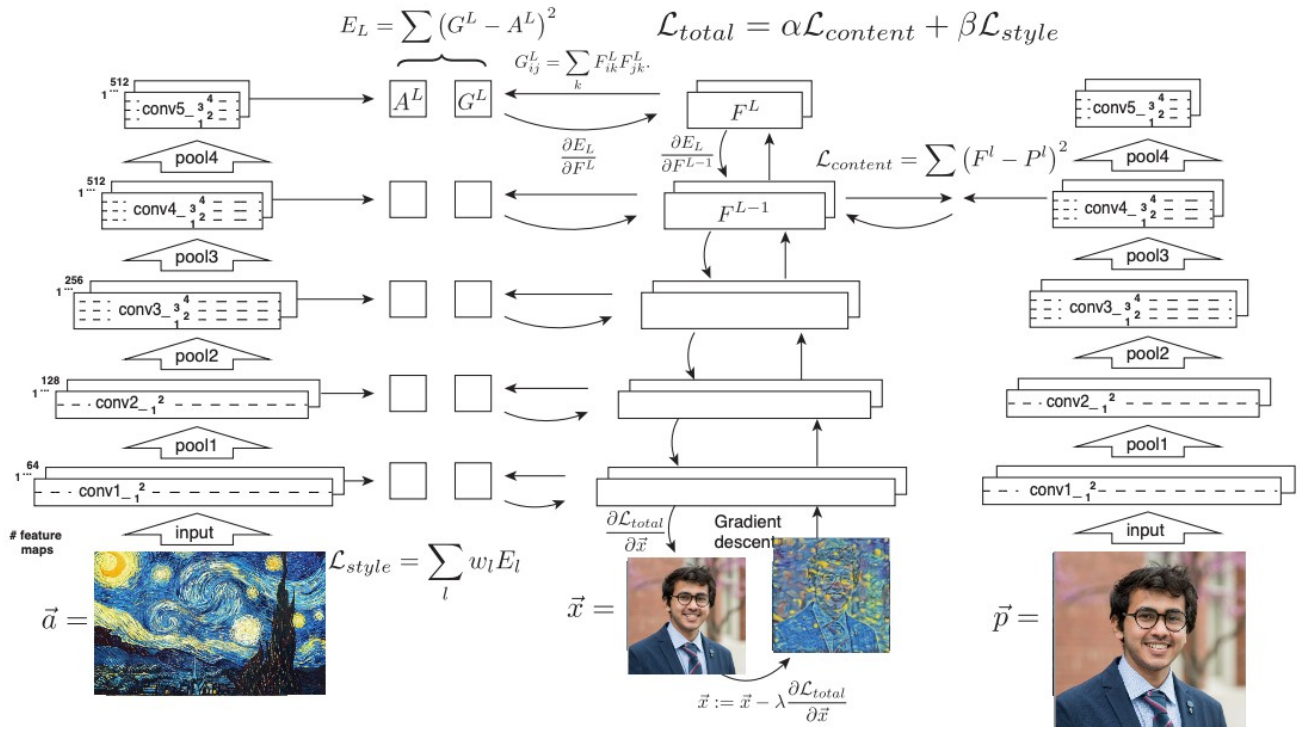
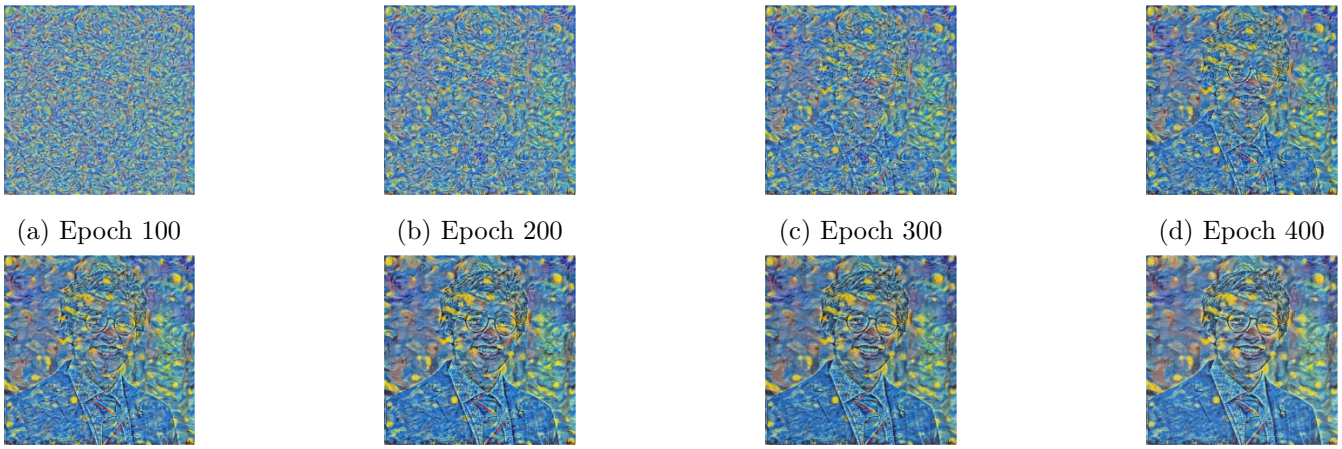


Figure 3: Style Transfer Algorithm of Gaty's research paper[1]



(a) Epoch 100 (b) Epoch 200 (c) Epoch 300 (d) Epoch 400

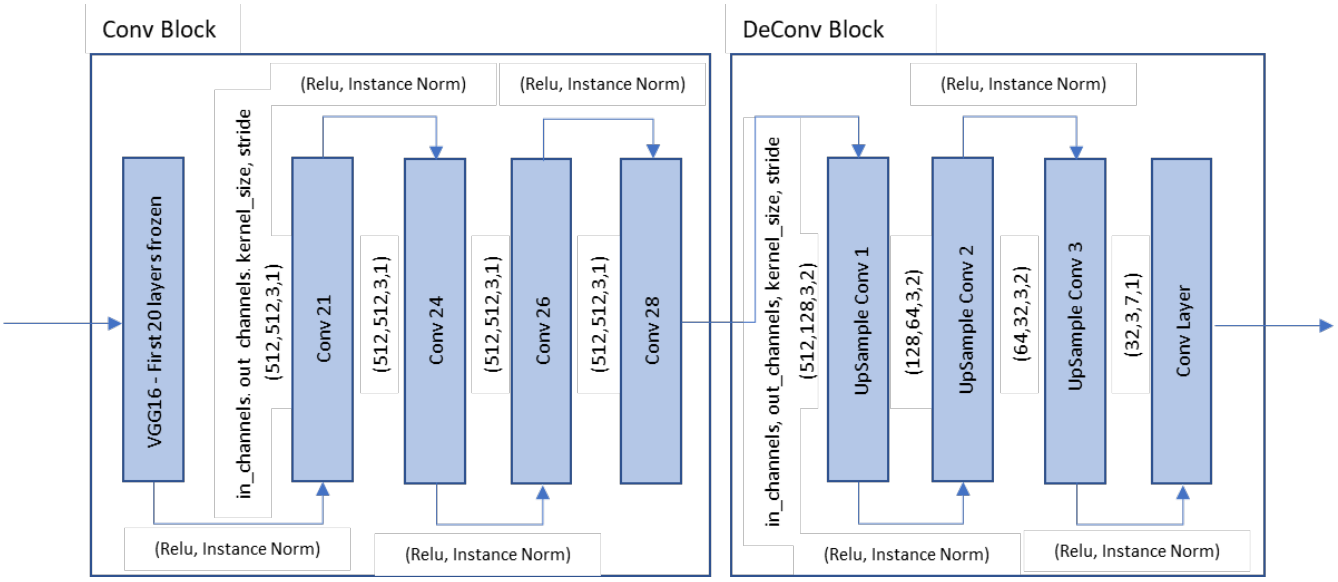
(e) Epoch 500 (f) Epoch 600 (g) Epoch 700 (h) Epoch 800

Figure 4: Progression of Style Transfer at various epochs using Gatys et al.'s approach

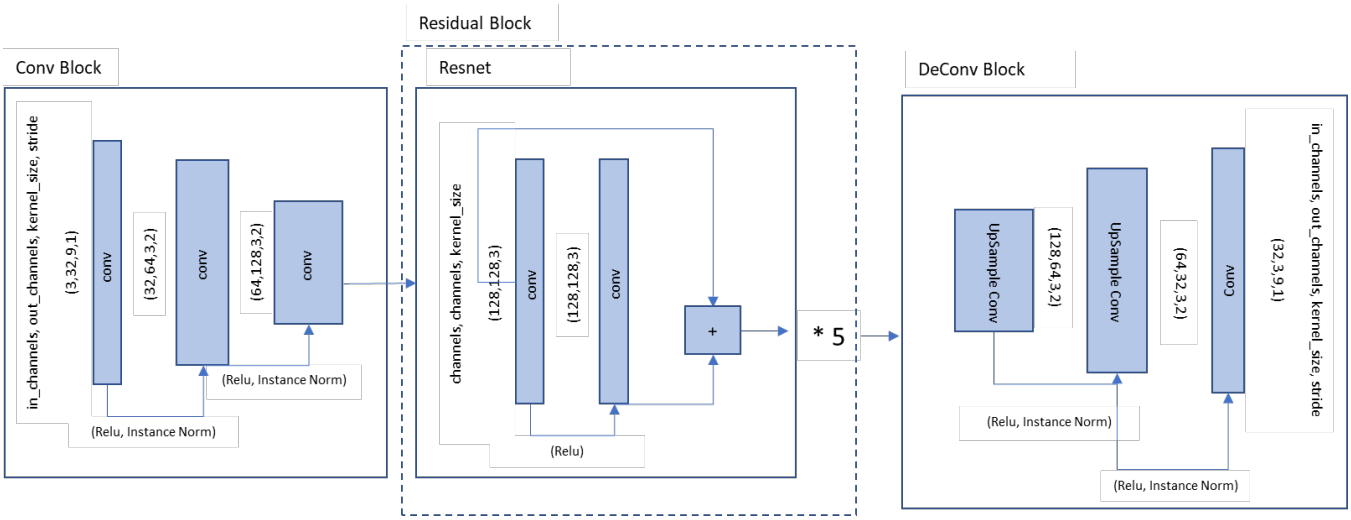
We wanted to compare the performance of the two model architectures. We chose VGG16's architecture as the encoder, and added upsampling layers that outputs the stylized image. The encoder decreases the spatial selectivity of the input image, while preserving the contextual information. The decoder increases the spatial selectivity of the output of the encoder, while reducing the output channels from 512 to 3. This

can be seen in Figure 5a.

We were highly inspired by the GoogleNet's aggressive down-sampling [8], and ResNet's idea of increasing the depth of the model using skip connections [9]. So, we aggresively downsampled in the initial layers of AONET by increasing the output channels from 3 to 128, and added five ResNet blocks of 128 channels. This is the encoder part of AONET. We decoded the output of the encoder using UpSampling layers or DeConv Block, where the output channels decreased from 128 to 3. This can be seen in Figure 5b.



(a) VGG16 with upsampling layes



(b) AONet with upsampling layes

Figure 5: VGG16 and AONet architectures with upsampling layers for Style Transfer

The algorithm for training the models is outlined in Algorithm 2. The algorithm is similar to the original paper's approach, with the only difference being that the model parameters are updated instead of the pixel values. We used pre-trained 16-layer VGG network[7] for the feature extraction.

For style representations, we chose all layers of the VGG16 model. We also computed the gram matrix of the style representations. Gram matrix A^l represents feature correlations, where A^l_{ij} is the inner product

between the vectorized feature maps i and j in layer l.

$$A_{ij}^{l} = \frac{(\sum_k S_{ik}^{l} S_{jk}^{l})}{(N_l x M_l x C)} \tag{4}$$

where C is channel.

For style transfer of style image, let's consider the training of model_aonet. For each epoch and mini batch, we passed the batches of content images through this model whose output is the generated image. So, we passed the batches of content images and the generated images through the feature extractor to get P and F as the feature representations of the batches of content images and generated images, respectively. We chose a desired content layer c (usually higher layer) to get us the P_c from any of the P_l content representations, because higher layers capture the high-level content in terms of objects. We then computed the content loss, as the mean squared error - similar to Algorithm 1.

$$\mathcal{L}_{content} = w_c * mse\_loss(P_c, F_c) \tag{5}$$

We then computed the gram matrix of the feature representations of the batches of generated images for all layers, and stored them in G.

The style loss is also defined as the mean squared-error loss between the gram matrices of the two style representations of the style image and the generated images:

$$\mathcal{L}_{style} = w_s * \sum_{l}^{alllayers} mse\_loss(A_l, G_l) \tag{6}$$

where A_l, G_l represent the gram matrix of the style representations of the style image and generated image for the layer l, and w_s is the style weight.

We then minimised total loss, which is the sum of the content loss and style loss. The loss that we minimized was the total loss $\mathcal{L}_{total}$. We then back-propagated and updated the parameters of the model such that we got the stylized image that reflects the semantic content of the content image and also preserves the style.

Before we generate the batches of content images from any of the datasets, all of these images are transformed in these sequential steps: Resize(TRAIN_IMAGE_SIZE), CenterCrop(TRAIN_IMAGE_SIZE), ToTensor(), Multiply by 255. We kept all the images to be of the same shape (TRAIN_IMAGE_SIZE = 256 or 226).

3.3 Image Processing Steps

We loaded the image using PIL in implementation of the original approach, and cv2 in implementation of training of our models (shown in Appendix). After loading and before sending the image to the model, we converted image to tensor by resizing it to the maximum of height or width, converting it to tensor and multiplying all values by 255 (shown in Appendix).

We converted the tensor of the output of the model to the image by converting it to the numpy array (shown in Appendix).

3.4 Memory of VGG16 encoder with upsamling layers

We observed that VGG16 encoder was too deep for our GPU training. It required more than 20 GB of training of all the layers. VGG16's pretrained initial layers capture the contextual information, while also reducing the spatial dimensions.

So, we applied the concept of transfer learning, where we froze the first 20 feature layers. However, the parameters of the remaining layers were updated. We replaced the MaxPool2d in the rest of the layers with Instance Norm layers [4]. This helped us reduce the GPU memory requirement to less than 16 GB.

Algorithm 2 Style Transfer with parameters of the model as trainable

Require:

Pretrained VGG16 model: $vgg\_fe()$ ▷ Feature Extractor

VGG16 or AONet Style Transfer: $model_st()$ as $vgg16()$ or $custom()$ ▷ Style Transfer Model

style_image, epochs, learning_rate, optimizer()

mse_loss() ▷ Mean Squared Error as Loss

gram_matrix() ▷ Gram Matrix Function to compute feature correlations

content_layer ▷ Desired Content Layer

w_c, w_s ▷ Content Weight, Style Weight, resp.

data ▷ Either COCO's Validation dataset or TinyImageNet

procedure STYLE_TRANSFER(vgg_fe, model_st, style_image, epochs, learning_rate, optimizer, mse_loss, style_layers, content_layer)

$S = [vgg\_fe(style\_image)[l]\ for\ all\ layers\ ]$ ▷ Get a list of Style Features of all layers of VGG16

$A = [gram\_matrix(S_l)\ for\ all\ layers\ ]$ ▷ Get list of target Gram Matrix of all layers of VGG16

$optim = optimizer([model\_st.parameters()], lr = learning\_rate)$ ▷ Initialize the optimizer

for each epoch **do**

for each content_batch in data **do**

$P_c = vgg\_fe(content\_batch)[content\_layer]$ ▷ Get Content Features for the desired content layer

$generated\_batch = model\_st(content\_batch)$ ▷ The output of the model_st is stylized image

$F = vgg\_fe(generated\_batch)$ ▷ Get features of generated image

$F_c = F[content\_layer]$ ▷ Get desired feature from desired content layer

$L_{content} = mse\_loss(P_c, F_c) * w_c$ ▷ Get Content loss, multiplied by Content Weight

$F_s = [F[l]\ for\ all\ layers\ ]$ ▷ list of desired features of generated image from all layers of VGG16 feature extractor

$G = [gram\_matrix(F_s[i])\ for\ i\ in\ len(F_s)]$ ▷ Get list of Generated Image's Gram Matrix

$L_{style} = 0$ ▷ Initialize style loss to 0

for each i in len(A) **do**

$L_{style} += mse\_loss(A[i], G[i]) * w_s$ ▷ Multiply by Style Weight

end for

$L_{total} = L_{content} + L_{style}$ ▷ Total loss

update the parameters of the model_st wrt L_{total}

end for

end for

save the model parameters for inference

end procedure

3.5 Web App Implementation

We experimented with two architectures, five styles, each trained on two different datasets during our experiments. As a result, we had twenty trained models for inference. We deployed all models as a web application to enable fast and easy comparison between these model types. To do that, we used Fast Dash, an open-source web application development framework created in Python. It leverages a pre-built user interface to build an interactive web application, drastically cutting down the development time. Fast Dash maps the user-specified input and output widgets with the inference function's input arguments and generates output. Our inference function has four inputs: the input image, model architecture, style, and dataset on which we trained the model.

Fast Dash uses a Flask backend to host web applications on a server. First, we developed and deployed a web application on our local machine to investigate the performance of our models and debug the application. Second, we containerized the application code using Docker. Finally, we uploaded the Docker image to Google Container Registry [10] and deployed it as a serverless container using Google Cloud Run [11]. The application is currently hosted on Google Cloud[1].

4 Results

We show results on image style transfer, where a feed-forward network is trained to solve the optimization problem in real-time. We used two datasets: **tinyimagenet** [2] and **COCO validation dataset**[3]. TinyImageNet dataset has lower quality images (64x64), while COCO validation dataset has higher quality images (640x480). We have also used the pretrained model parameters of VGG16 model from EECS department, UMich [12].

4.1 Hyperparameter Tuning

We played with the following hyper-parameters for getting better results. We have trained 20 models corresponding to 2 model architectures (VGG16_model), 2 datasets (TinyImageNet and COCO Validation) and 5 styles (Rain Princess, The Scream, Udnie, Wave, The Shipwreck). The hyperparameter settings are outlined in Table 1. We have outlined the following hyperparameters, namely, train_image_size, number of epochs, batch size, content weight, style weight, learning rate and content layer.

Our goal was to train a model that will output the stylized image with balanced trade-off between content and style matching, explained more in 4.1.1.

We resized the images to **TRAIN_IMAGE_SIZE** of 256 in most models, except for (VGG16-TinyImageNet-RainPrincess), where we kept it as 224, highlighted in Table 1.

We kept the number of epochs as 1. We only required one iteration of the whole training to get the stylized image. We kept a small batch size of 4 , since these models require 20 GB of GPU memory for training. The learning rate was kept constant at $1e{-}3$ for all models' training.

4.1.1 Trade-off between content and style matching

While we synthesized the image that combines the content of one image with the style of another image, there would come a time when we don't get an image that matches both constraints at the same time. While the original approach method in Gatys et al's research paper [1] used content weight of 100 and the style weights of $1e7/n^2$ for n in $[64, 128, 256, 512, 512]$ for each chosen style layer, the training of VGG16 and AONet used only one content weight and style weight. Gatys et al. method's emphasis on style resulted in images that match the appearance of the network, effectively giving us a texturised version of it, but show hardly any content, as can be seen in Figure 1. If we wanted more emphasis on the content, we could decrease the style weight. So, we played with various content weight-style weight combinations, and came up with the best combination for our models. Content weight is 17 (1, resp.) and style weight is 50 (5,

[1] `https://web-app-style-transfer-hpn4y2dvda-uc.a.run.app/`

resp.) for 15 (5, resp.) out of 20 trained models. The resulting stylized images for (VGG16-TinyImageNet) models with (17, 50) as (w_c, w_s) had a lot of noise.

4.2 Qualitative Examples

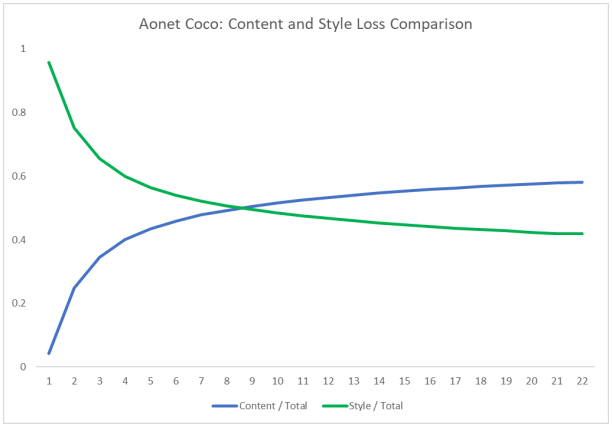
We present some of the best stylized images for all five styles in Figure 8. The quality of our stylized images is quite amazing, as compared to our Gatys et al's resulting images in Figure 1. Compared with the pixel-based update training method, our method appears to transfer the style more accurately for most images. Our best model is AONET trained on COCO's validation dataset.

Model	**Dataset**	**Style**	**TRAIN _IMAGE _SIZE**	**NUM _ EPOCHS**	**BATCH _SIZE**	**W_C**	**W_S**	**LR**	**CONTENT _LAYER**
AONET	COCO Val	Rain Princess	256	1	4	17	50	0.001	relu2_2
		The Scream	256	1	4	17	50	0.001	relu2_2
		The Ship wreck	256	1	4	17	50	0.001	relu2_2
		Udnie	256	1	4	17	50	0.001	relu2_2
		Wave	256	1	4	17	50	0.001	relu2_2
	Tiny Image Net	Rain Princess	256	1	4	17	50	0.001	relu1_2
		The Scream	256	1	4	17	50	0.001	relu1_2
		The Ship wreck	256	1	4	17	50	0.001	relu1_2
		Udnie	256	1	4	17	50	0.001	relu1_2
		Wave	256	1	4	17	50	0.001	relu1_2
VGG16	COCO Val	Rain Princess	256	1	4	17	50	0.001	relu2_2
		The Scream	256	1	4	17	50	0.001	relu2_2
		The Ship wreck	256	1	4	17	50	0.001	relu2_2
		Udnie	256	1	4	17	50	0.001	relu2_2
		Wave	256	1	4	17	50	0.001	relu2_2
	Tiny Image Net	Rain Princess	**224**	1	4	**1**	**5**	0.001	relu1_2
		The Scream	256	1	4	**1**	**5**	0.001	relu1_2
		The Ship wreck	256	1	4	**1**	**5**	0.001	relu1_2
		Udnie	256	1	4	**1**	**5**	0.001	relu1_2
		Wave	256	1	4	**1**	**5**	0.001	relu1_2

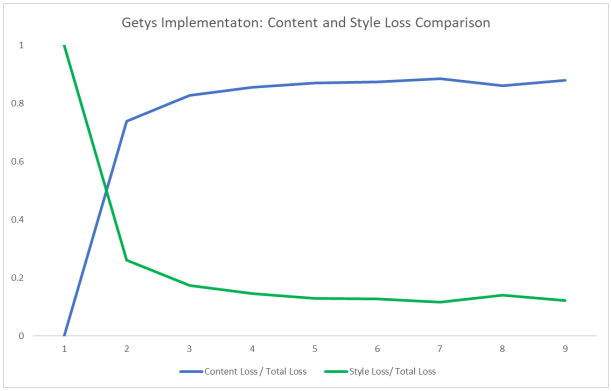
Table 1: Hyper-parameter settings that worked for the trained models

4.3 Quantitative Evaluation

The quality of our models according to Algorithm 2 was better than the original approach outlined in Algorithm 1. To evaluate this quantitatively, we compared the ratio of content and style losses (with the total loss) for the two approaches. The two charts below highlight our findings.



(a) Loss Ratios for AONET trained on COCO



(b) Loss Ratios Getys et al approach

Figure 6: Loss ratios for the model and pixel based methods

We observed that since the gap between the content and style loss ratios was very less for the model based method, it was able to preserve the structure and transfer the style effectively. This finding was in contrast to the pixel based method implemented by Gatys et al. [1] wherein the gap between the content loss ratio and the style loss ratio was more than what we found in model based training. Owing to this, the structure of the output image was lost. The tables that were used to generate the two figures in this section can be found in Appendix A.

4.4 Speed of Training

All of the results are obtained on **NVIDIA TESLA V100 SXM2 16GB**. In Table 2, we compare the speeds of all models with Gatys et al's approach. At inference, Gatys et al.'s approach took about 480 seconds to stylize the image, since the users' pixel values were updated. Our method took about $2.5 - 3$ seconds to generate the stylized image, since we had saved the model parameters. The inference is done on CPU, since we wanted to compare the web application's processing time.

Model	**Dataset**	**Style**	**Training Time for last batch (sec)**
AONET	**COCO Val**	**Rain Princess**	1532
		The Scream	1418
		The Shipwreck	1409
		Udnie	1673
		Wave	1560
	TinyImageNet	**Rain Princess**	1732
		The Scream	1093
		The Shipwreck	1784
		Udnie	2257

Model	Dataset	Style	Training Time for last batch (sec)
		Wave	1875
VGG16	**COCO Val**	**Rain Princess**	831
		The Scream	742
		The Shipwreck	775
		Udnie	918
		Wave	1110
	TinyImageNet	**Rain Princess**	1294
		The Scream	1258
		The Shipwreck	1313
		Udnie	1497
		Wave	1624

Table 2: Time for training (in seconds) for 256 x 256 images (except for VGG16, COCO, Rain Princess) compared to Gatys et al.'s training speed of 180 seconds (on GPU)

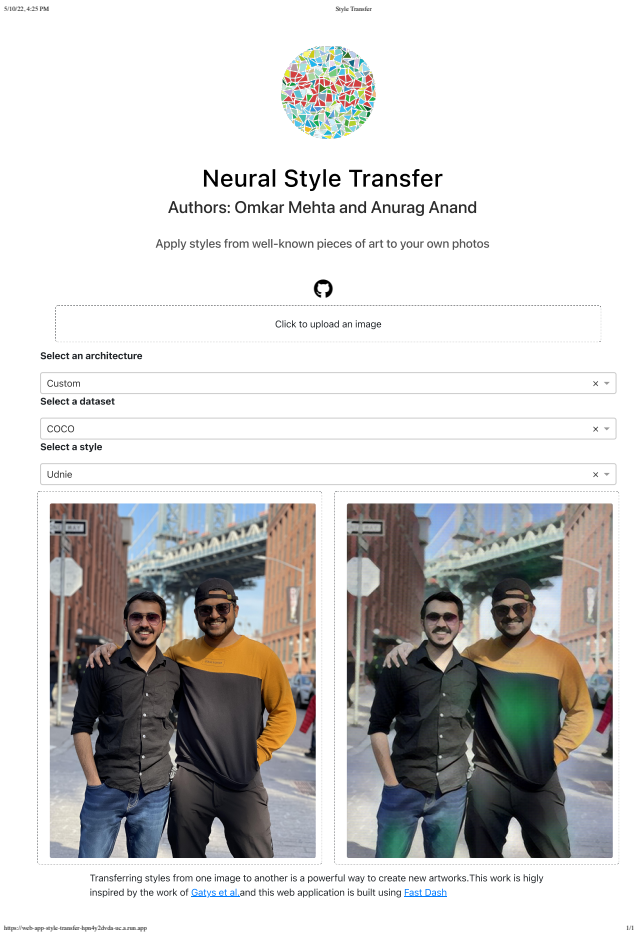


Figure 7: Layout of the Web Application, deployed on Google Cloud Run

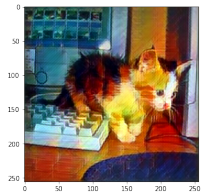
(a) Rain Princess

(b) The Scream

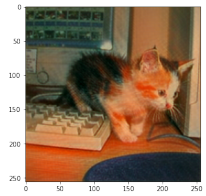
(c) The Shipwreck

(d) Udnie

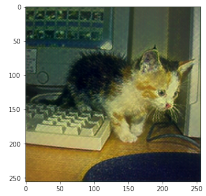
(e) Wave



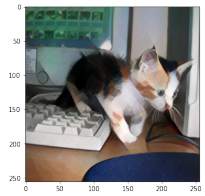
(f) AONet, COCO, Rain Princess



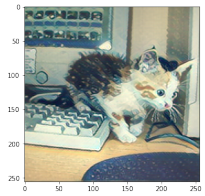
(g) AONet, COCO, Scream



(h) AONet, COCO, Shipwreck



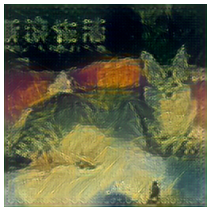
(i) AONet, COCO, Udnie



(j) AONet, COCO, Wave

(k) AONet, TinyImageNet, Rain Princess

(l) AONet, TinyImageNet, Scream

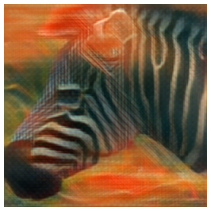


(m) AONet, TinyImageNet, Shipwreck

(n) AONet, TinyImageNet, Udnie

(o) AONet, TinyImageNet, Wave

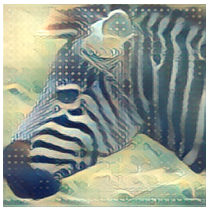
(p) VGG16, COCO, Rain Princess



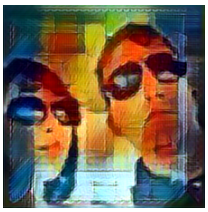
(q) VGG16, COCO, Scream

(r) VGG16, COCO, Shipwreck

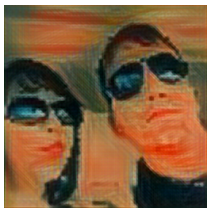
(s) VGG16, COCO, Udnie



(t) VGG16, COCO, Wave

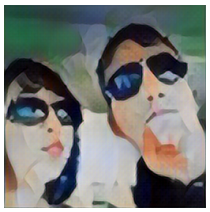


(u) VGG16, TinyImageNet, Rain Princess



(v) VGG16, TinyImageNet, Scream

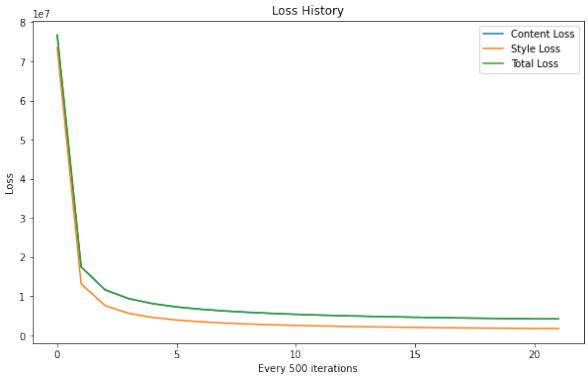
(w) VGG16, TinyImageNet, Shipwreck



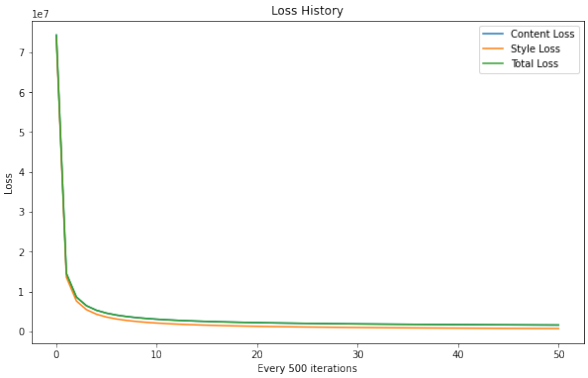
(x) VGG16, TinyImageNet, Udnie

(y) VGG16, TinyImageNet, Wave

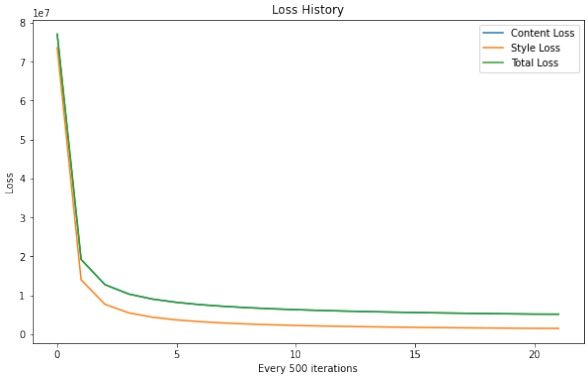
Figure 8: Batch Images Stylized with Different Models trained with different styles on two datasets



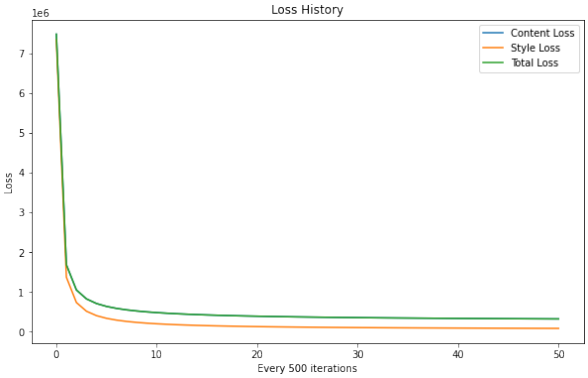
(a) AONet, COCO, Rain Princess



(b) AONet, TinyImageNet, Rain Princess



(c) VGG16, COCO, Rain Princess



(d) VGG16, TinyImageNet, Rain Princess

Figure 9: Content Loss, Style Loss and Total Loss with Different Models trained with Rain Princess style on two datasets

4.5 Results with different datasets

Since the COCO dataset had high quality content images, the outputs of the models trained on this dataset had their content quality preserved. Since the TinyImageNet dataset had low quality images (64x64), the outputs of the model looked more animated. We didn't change the content and style weights for the models trained on TinyImageNet, because we were happy with the animated results. This can be seen in Figure 8k.

4.6 Inference with Web Application

The working web application's layout is shown in Figure 7. It takes about 2-3 seconds, based on the user image's quality. Since we had limited resources (we have allocated only 8 GB of memory and 4 CPUs to each container instance), we resize all user images to $256x256$ before passing it to the chosen model.

5 Discussion and Conclusion

Gatys et al. introduced a neural algorithm that renders a content image in the style of another image, achieving so-called style transfer. However, their framework requires a slow iterative optimization process, which limits its practical application like fast inference at test time. Fast approximations with feed-forward neural networks like ours sped up neural style transfer, by directly updating the parameters of the model. Beyond the fascinating applications like web application where the user can upload the photo and get the stylized image in a few minutes, we have got a fair understanding of deep image representations. Gatys et al. [1] employed an optimization-based process to manipulate pixel values to match feature statistics.

We replaced that process with neural networks having encoders, instance normalization [4] and decoder (upsampling layers) that directly aligns the feature statistics in one shot and inverts the features back in the pixel space.

In future work, since there is room for improvement, we want to add more skip connections to our architecture. Also, right now, we trained the model pertaining to only one style image. This induced a slight bias leading to the issues in generalisation of the model. So, we have planned to train the model on a set of different styles at the same time so that the models are style-invariant.

We also want to deploy a web application where the user can upload the video and get the stylized version of the video in real time.

All models are stored in Dropbox [2]. All the code can be found in our GitHub repository. [3].

6 Statement of individual contribution

We divided our work in equal proportions where we could. While Omkar trained the models of all styles pertaining to AONet with COCO and VGGNet with TinyImageNet, Anurag trained the models of all styles pertaining to AONet with TinyImageNet and VGGNet with COCO. We developed the fast dash application together in one sitting. The discussions, analysis and literature survey was done prior to actually finalizing the approach and implementing it from scratch.

References

[1] L. A. Gatys, A. S. Ecker, and M. Bethge, "Image style transfer using convolutional neural networks," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 2414–2423.

[2] "Tinyimagenet dataset," http://cs231n.stanford.edu/tiny-imagenet-200.zip.

[3] "Validation data of coco dataset," http://images.cocodataset.org/zips/val2014.zip.

[4] D. Ulyanov, A. Vedaldi, and V. Lempitsky, "Instance normalization: The missing ingredient for fast stylization," 2016. [Online]. Available: https://arxiv.org/abs/1607.08022

[5] W. T. F. Alexei A. Efros, "Image quilting for texture synthesis and transfer," in *01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 2001, pp. 341–346.

[6] A. Hertzmann, C. E. Jacobs, N. Oliver, B. Curless, and D. H. Salesin, "Image analogies," in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 327–340. [Online]. Available: https://doi.org/10.1145/383259.383295

[7] "Vgg-16 pre-trained model from pytorch," https://pytorch.org/hub/pytorch_vision_vgg/.

[8] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Computer Vision and Pattern Recognition (CVPR)*, 2015. [Online]. Available: http://arxiv.org/abs/1409.4842

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[10] "Google container registry," https://cloud.google.com/container-registry.

[11] "Google cloud run," https://cloud.google.com/run.

[2] `https://www.dropbox.com/s/jb5lq0bgynyz3wm/models.zip?dl=0`

[3] `https://github.com/OmkarMehta/style_transfer_webapp.git`

[12] “Vgg-16 pre-trained model parameters from eecs, umich,” https://web.eecs.umich.edu/~justincj/models/vgg16-00b39a1b.pth.

A Table for all losses in Gatys et al. implementation and our model's implementation

Loss Type/Epoch	0	100	200	300	400	500	600	700	800
Content Loss	4	300.2	268.3	246.7	232.1	221.8	216	215.8	209.9
Style Loss	9556	106.3	56.1	41.9	34.5	32.2	28.3	35	29.1
Total Loss	9560	406.5	324.4	288.6	266.6	254	244.3	250.8	239

Table 3: Content Loss, Style Loss, and Total Loss of Gatys et al.'s approach

# Batches	Content Loss	Style Loss	Total Loss
1	3.21E+06	7.35E+07	7.67E+07
2	4.35E+06	1.32E+07	1.75E+07
3	4.02E+06	7.64E+06	1.17E+07
4	3.75E+06	5.63E+06	9.38E+06
5	3.53E+06	4.59E+06	8.12E+06
6	3.34E+06	3.94E+06	7.28E+06
7	3.20E+06	3.49E+06	6.69E+06
8	3.08E+06	3.17E+06	6.25E+06
9	2.98E+06	2.92E+06	5.91E+06
10	2.90E+06	2.72E+06	5.63E+06
11	2.83E+06	2.56E+06	5.39E+06
12	2.77E+06	2.43E+06	5.20E+06
13	2.72E+06	2.31E+06	5.03E+06
14	2.67E+06	2.21E+06	4.89E+06
15	2.63E+06	2.13E+06	4.76E+06
16	2.59E+06	2.05E+06	4.64E+06
17	2.56E+06	1.98E+06	4.54E+06
18	2.53E+06	1.93E+06	4.45E+06
19	2.50E+06	1.87E+06	4.37E+06
20	2.48E+06	1.82E+06	4.30E+06
21	2.45E+06	1.78E+06	4.23E+06
22	2.45E+06	1.77E+06	4.21E+06

Table 4: Content Loss, Style Loss, and Total Loss of the AONET model trained on coco dataset