Artistic Style Transfer Implementation with Tensorflow

GR5242 Advanced Machine Learning Final Project

Tian Gao
tg2585
tian.gao@columbia.edu

November 2, 2017

Contents

1 Introduction **1**
1.1 Project background . . . 1
1.2 Related works . . . 1
1.3 Development environment and technical specifications . . . 2

2 Mathematical Foundation **3**
2.1 Covolutional neural network and rectified linear units . . . 3
2.2 VGG-network . . . 3
2.3 Loss function . . . 3
2.3.1 Content loss . . . 4
2.3.2 Style loss . . . 4
2.3.3 Total variation loss . . . 4

3 Tensorflow Implementation **5**
3.1 System overview . . . 5
3.2 Pre-defined parameters . . . 5

4 Sample Results **7**
4.1 When New York meets Van Gogh . . . 7
4.1.1 Checkpoint output . . . 7
4.1.2 Loss convergence . . . 9
4.2 More examples . . . 10

5 Discussion **12**
5.1 Tuning parameters optimization . . . 12
5.1.1 Loss weighting . . . 12
5.1.2 Convolutional layer selection . . . 12
5.1.3 Gradient descent parameters . . . 12
5.2 Run time optimization . . . 12

A Code Manuscript **13**

B Python Code **14**
B.1 style_transfer.py . . . 14
B.2 neural_network.py . . . 16
B.3 visual_geometry_group.py . . . 19

List of Figures

1–1 An image of a cat rendered via Prisma 1

3–1 Tensorflow implementation system overview 5

4–1 Raw input of New York photo and Van Gogh painting 7
4–2 Overlay Starry Night on New York 7
4–3 Raw input of New York photo and Van Gogh painting 8
4–4 Model training loss plot 9
4–5 Model training log loss plot 10
4–6 Overlay Picasso on New York 11
4–7 Overlay Pollock on New York 11

List of Tables

1–1 Technical specification 2

3–1 Pre-defined model parameters 6
3–2 Pre-defined feature extraction layers 6

A–1 Project directory 13

1 Introduction

1.1 Project background

In 2016, a mobile application named Prisma was launched [1] and became popular rapidly. The app is attractive to many people because it enables the user to transform your own photo into artistics effect; e.g. make a photo of your cat look like a painting [1].

Figure 1–1: An image of a cat rendered via Prisma

The photo-editing app utilizes convolutional neural network techniques to extract the content and style of two graphs, and then transfer the artistic style from one to the other, in order to combine the content of a photo with the style of another.

The artistic style trasfer algorithm was initially proposed in [2] and then published as [3]. This project is intended to implement the algorithm described in [3] (hereinafter referred to as "the original work") and build a Python-Tensorflow based artistic style transfer system (hereinafter referred to as "the system").

1.2 Related works

The topic of artistic transfer has been popular since 2015. There have been a lot of excellent works and implementations done in the area. The Prisma app would be a good example.

The authors of [3] founded a website named `DeepArt.io` themselves [4], which provides an access for users to upload content and style images.

Another excellent implementation inspired by the original work is by Anish Athalye on GitHub [5]. The work is also Tensorflow-based but has more complicated command line parsers than does this work. Some Python code is borrowed from Anish's work with extra re-factoring, re-structuring, and re-encapsulation.

For content representation (content feature extraction) part, some works have been done in the area of semantic segmentation with convolutional neural network, among which [6] is a good example and referred by the original work.

For style representation (style feature extraction) part, related works focus on generative networks and texture synthesis [7, 8].

1.3 Development environment and technical specifications

The Python-Tensorflow based artistic style transfer system is built with Python3.6 and Tensorflow1.3.0. This information along with other Python libraries required to run the code are all recorded in `requirements.txt`. See Appendix A for details.

All the examples are generated with a 2012 Macbook Pro laptop. Table 1–1 shows some specifications of the laptop, mainly hardware specs like computing power.

hardware parameter	technical specifications
processer (CPU)	2.9GHz, dual-core Intel Core i7
memory	8 GB 1600 MHz DDR3
graphics (GPU, un-used)	Intel HD Graphics 4000 1024 MB

Table 1–1: Technical specification

2 Mathematical Foundation

2.1 Covolutional neural network and rectified linear units

One of the most popular topics nowadays in deep learning area is visual perception, which includes face recognition, visual cognition, etc. Among all sorts of vision models, deep neural network models, especially convolutional neural networks (CNN), are the most commonly used with due to models' high perceptual quality [2, 9, 10].

Basic principles of CNN are already introduced in class and a lot of other research papers thus we will not expand the discussion in this work.

One thing worth mentioning is that there has been a lot of works discussing about activation function selection. Common choices include sigmoid function, softmax function (higher dimensional, comparable to sigmoid), and one of the most popular, ReLU (rectified linear units). Introduced in [11] and [12], ReLU helps speeding training process and preserves information while going through layers. This property is favorable especially when training large scale neural network, and help improve model quality in areas where information preservation is significant, like visual perception.

Similarly, in downsampling mechanisms, different pooling methods may lead to various focus and thus performance. In this project we set default pooling method as max pooling, while average pooling is also common to use and can be set with certain changes in the code.

2.2 VGG-network

The most time-consuming part of building a neural network lies in model training/fitting stage. In order to fit a neural network, one needs huge amount of training data and validation data, with high-performance computing units and even parallel computation frameworks, which is impossible to implement within a few hours on a personal laptop.

Therefore, like the original work, we utilize a pre-trained neural network model by the Visual Geometry Group (VGG). In [13] the authours investigated the impact of CNN depth on large-scale image recognition, and they trained a few networks, with number of layers ranging from 16 to 19. We utilize the feature space provided by the VGG-network on [14].

2.3 Loss function

The total loss consists of three components: content loss, style loss, and total variation loss. Of the three, content loss and style loss are also known as pixel loss since the Euclidean distances are calculated pixel-wise.

The loss function has the form

$$L_{\text{total}} = \alpha L_{\text{content}} + \beta L_{\text{style}} + \gamma L_{\text{tv}}. \tag{2–1}$$

Different choices of the weights for each loss component will lead to different output; more commonly, larger α/β ratio helps the output graph resemble the original content image more while smaller α/β ratio emphasizes more on style representation.

In this work we choose $\alpha/\beta = 10^{-2}$. More details are discussed in Sec. 3.2 and Sec. 5.1.

2.3.1 Content loss

The content loss measures the ℓ_2 loss between output graph and the input content image:

$$L_{\text{content}}(x, p, l) = \frac{1}{|l|}\sum_{i,j}(x_{ij}^l - p_{ij}^l)^2, \tag{2–2}$$

where p_{ij} stands for the original image's feature representation in layer l and x_{ij} for the output image, and $|l|$ represents the size of the image representation at the layer; that is, the content loss function is a normalized Euclidean distance between the original image representation and the output image.

2.3.2 Style loss

The style loss measures the ℓ_2 loss between output graph and the input style image but has a more complicated form than of content loss: a Gram matrix is constructed to represent the feature correlations through inner product between feature map

$$\begin{aligned} G^l(x) &= F^l \cdot F^{l\prime}, \\ E^l(x, a) &= \frac{1}{|l|}\sum_{i,j}(G_{ij}^l(x) - a_{ij}^l)^2, \\ L_{\text{style}}(x, a; w) &= \sum_l w_l E^l, \end{aligned} \tag{2–3}$$

where F is the feature map matrix, a_{ij}^l represents the style image representation at layer l, E^l represents the style loss at layer l and w_l for loss weight for each layer.

Similar to content loss, the loss on each layer is a normalized Euclidean distance between the style representation and output image, and total style loss is the weighted average of individual losses. By default we set $w_l = \frac{1}{L}$, where L is the number of layers used for style feature extraction.

2.3.3 Total variation loss

The total variation loss is not a concept proposed in [3]. In [15] total variation loss is added as a regularization method in favor of spatial smoothness. This is also a common technique used in super-resolution works [16, 17].

The calculation of total variation loss can be represented as

$$L_{\text{tv}} = \sum_i^{d_1}\sum_j^{d_2}\left[\frac{1}{d_1}(x_{i+1,j} - x_{i,j})^2 + \frac{1}{d_2}(x_{i,j+1} - x_{i,j})^2\right]. \tag{2–4}$$

Essentially the total variation loss is also a type of ℓ_2 loss, i.e. Euclidean distance, but between pixels of the image itself so as to measure the overall smoothness.

3 Tensorflow Implementation

3.1 System overview

The style transfer system is implemented mainly with Python Tensorflow, with auxiliary packages including NumPy, SciPy, and Pillows. A flowchart of the system overview is provided in Fig. 3–1.

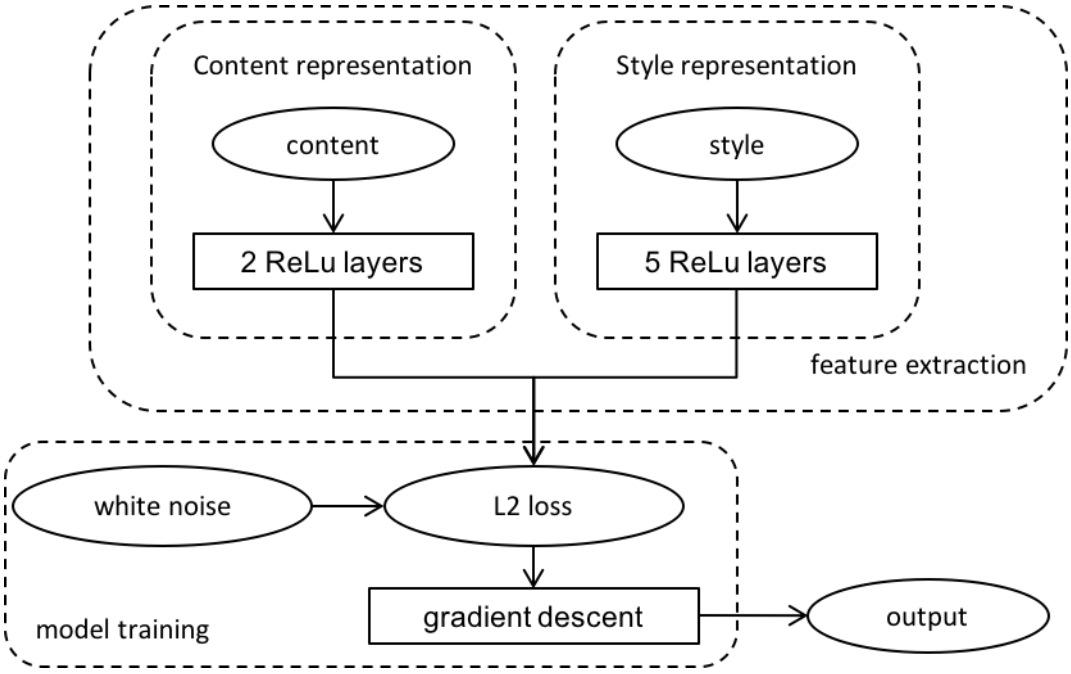


Figure 3–1: Tensorflow implementation system overview

In feature extraction part, the pre-trained VGG network is used for both content representation and style representation. One ReLU layers (relu4_2) is for content feature extraction and five ReLU layers (relu1_1, relu2_1, relu3_1, relu4_1, relu5_1) are for stye representation, which are different from the original work [3].

In model training part, the final output is initialized with a white noise picture, and a gradient-based optimization methodology known as Adam is used. As is described in [18], Adam optimizer is based on first-order gradient. Other optimizers including traditional gradient descent method can also be implemented.

3.2 Pre-defined parameters

Some tuning parameters are hard-wired in the system. Most of the choices follow the original work [3], or the reference [5] and some are modified.

There is plenty of room with cross-validatio optimization for performace improvement (see Sec 5.1 for more details). For simplicity, they are fixed in this system for now and the user can easily modify part of them in `constants.py` following instruction in A.

Some pre-defined parameters are listed below in Table 3–1. The ratio of content loss and style loss is arbitrarily set while [3] explores different loss ratios and corresponding output results. Parameters can always be modified and optimized with techniques like cross-validation.

parameter type	parameter value
content weight	5
style weight	500
total variation weight	100
Adam learning rate	10
Adam β_1	0.9
Adam β_2	0.999
Adam ϵ	10^{-8}
maximum iteration	1000
pooling layer method	max

Table 3–1: Pre-defined model parameters

Apart from model parameters, neural network choice in this work is also different.

		layers used
content representation	this work	relu4_2
	original work	conv1_1, conv4_2
	Anish work	relu4_2, relu5_2 (weight adjustable)
style representation	this work	relu1_1, relu2_1, relu3_1, relu4_1, relu5_1
	original work	conv1_1, conv2_1, conv3_1, conv4_1, conv5_1
	Anish work	relu1_1, relu2_1, relu3_1, relu4_1, relu5_1

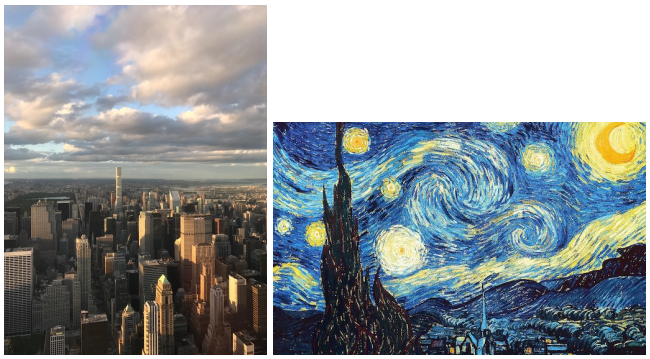
Table 3–2: Pre-defined feature extraction layers

Specifically, all layers in style representation are equally weighted. The random initialization, i.e. process to generate the white noise graph is arbitrary as well but has little impact on the output results, and thus is omitted here.

4 Sample Results

4.1 When New York meets Van Gogh

The first sample output here is overlay Van Gogh's starry night[1] on a picture of New York[2].



(a) New York photo (b) Van Gogh Starry Night

Figure 4–1: Raw input of New York photo and Van Gogh painting

With default parameters specified in Sec. 3.2, we have the output graph below as Fig. 4–2

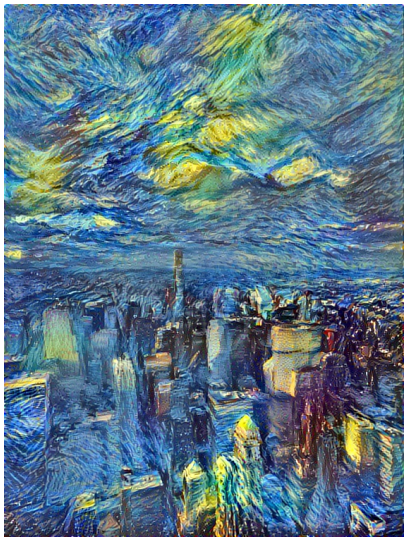
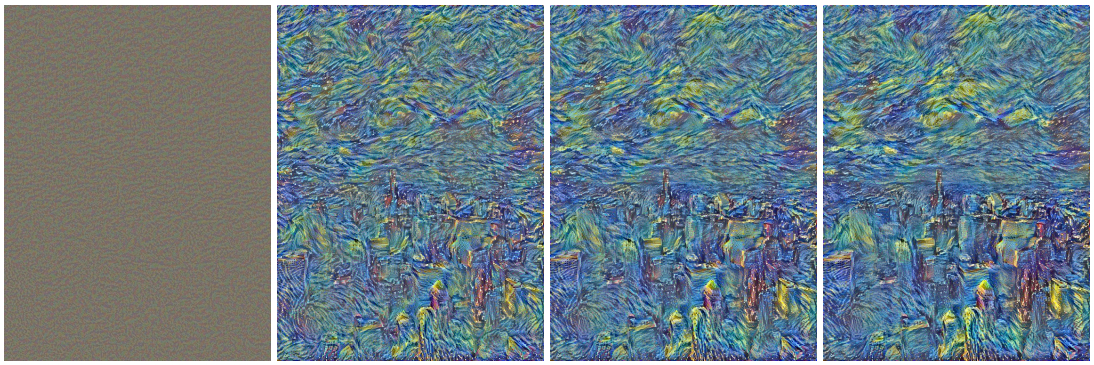


Figure 4–2: Overlay Starry Night on New York

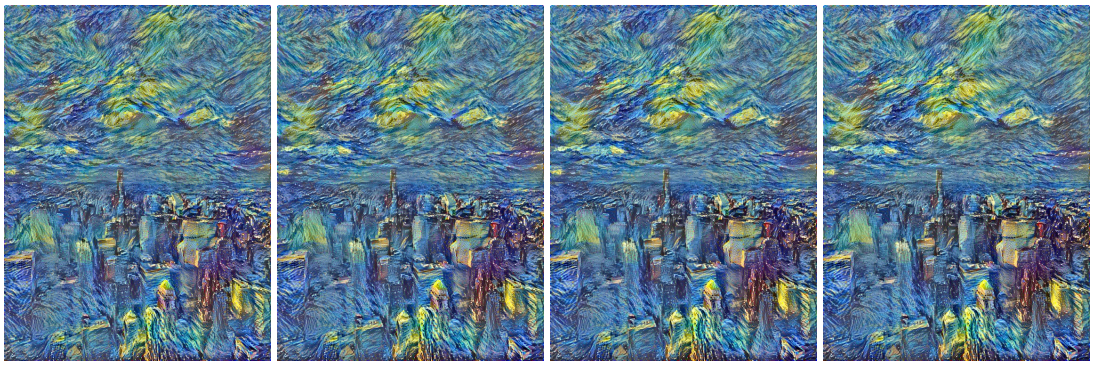
4.1.1 Checkpoint output

By default, we set number of iteration to 1000, and save intermediate results every 100 steps. In Fig. 4–3, we list the output image at each checkpoint N, including the initial white noise graph.

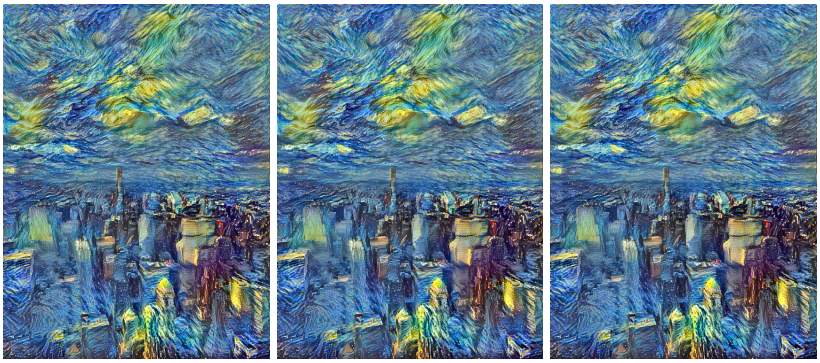
[1] Vincent Van Gogh, Starry Night (1889)
[2] The author took this photo in May 2017



(a) $N = 0$, white noise (b) $N = 100$ (c) $N = 200$ (d) $N = 300$



(e) $N = 400$ (f) $N = 500$ (g) $N = 600$ (h) $N = 700$



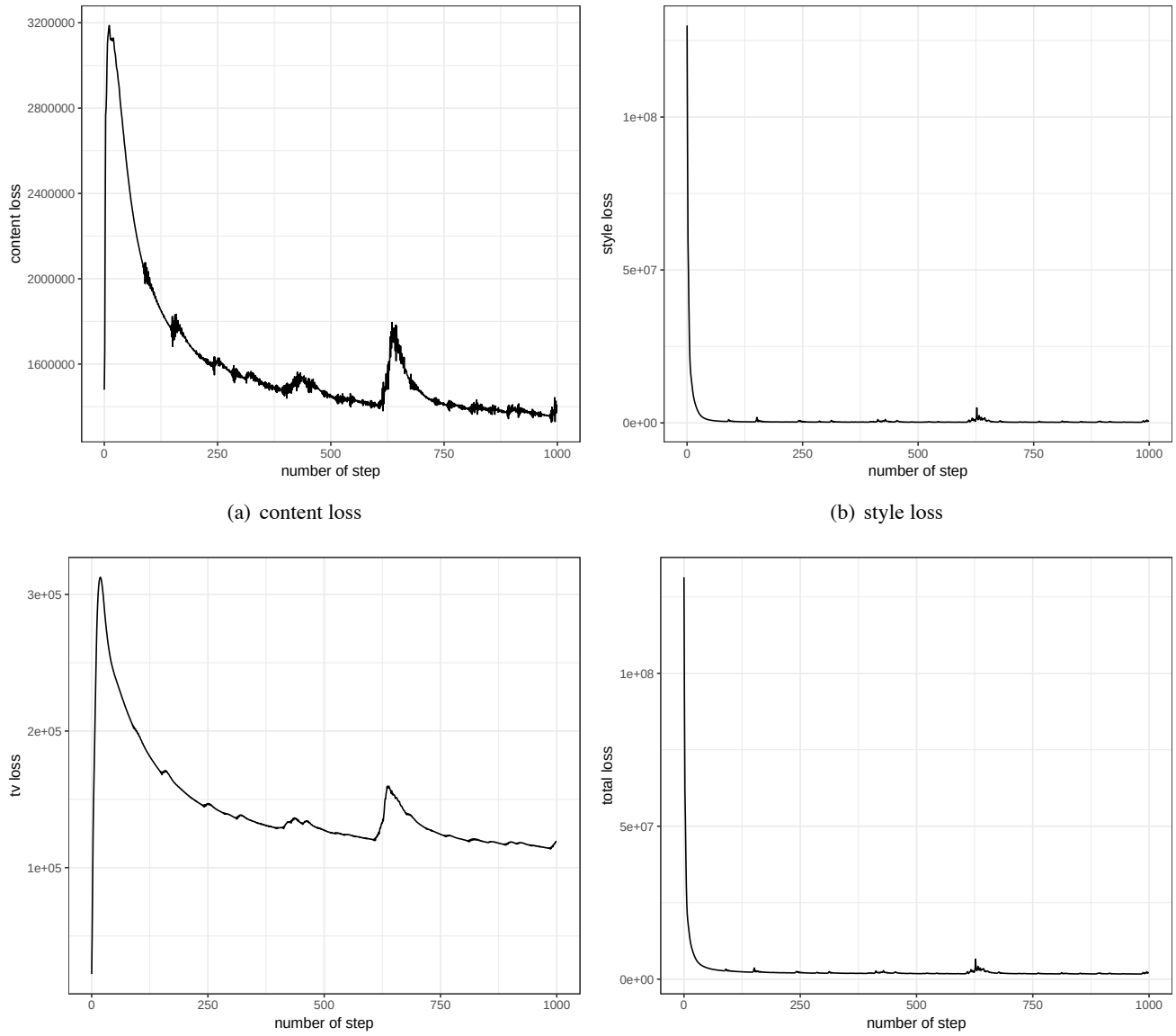
(i) $N = 800$ (j) $N = 900$ (k) $N = 1000$, final

Figure 4–3: Raw input of New York photo and Van Gogh painting

As is shown from the illustration above, the initial learning rate of the algorithm is fast and then the speed slows down; the output result has almost converged after $N = 500$. This pattern fits the common process of other deep learning algorithms.

4.1.2 Loss convergence

As is pointed out in Sec. 4.1.1, total loss declines rapidly at initial stage and then converges around $N = 600$. Fig. 4–4 shows the loss function values at each iteration. The overall concave shape of total loss can be viewed as a proof of the finding.



(a) content loss

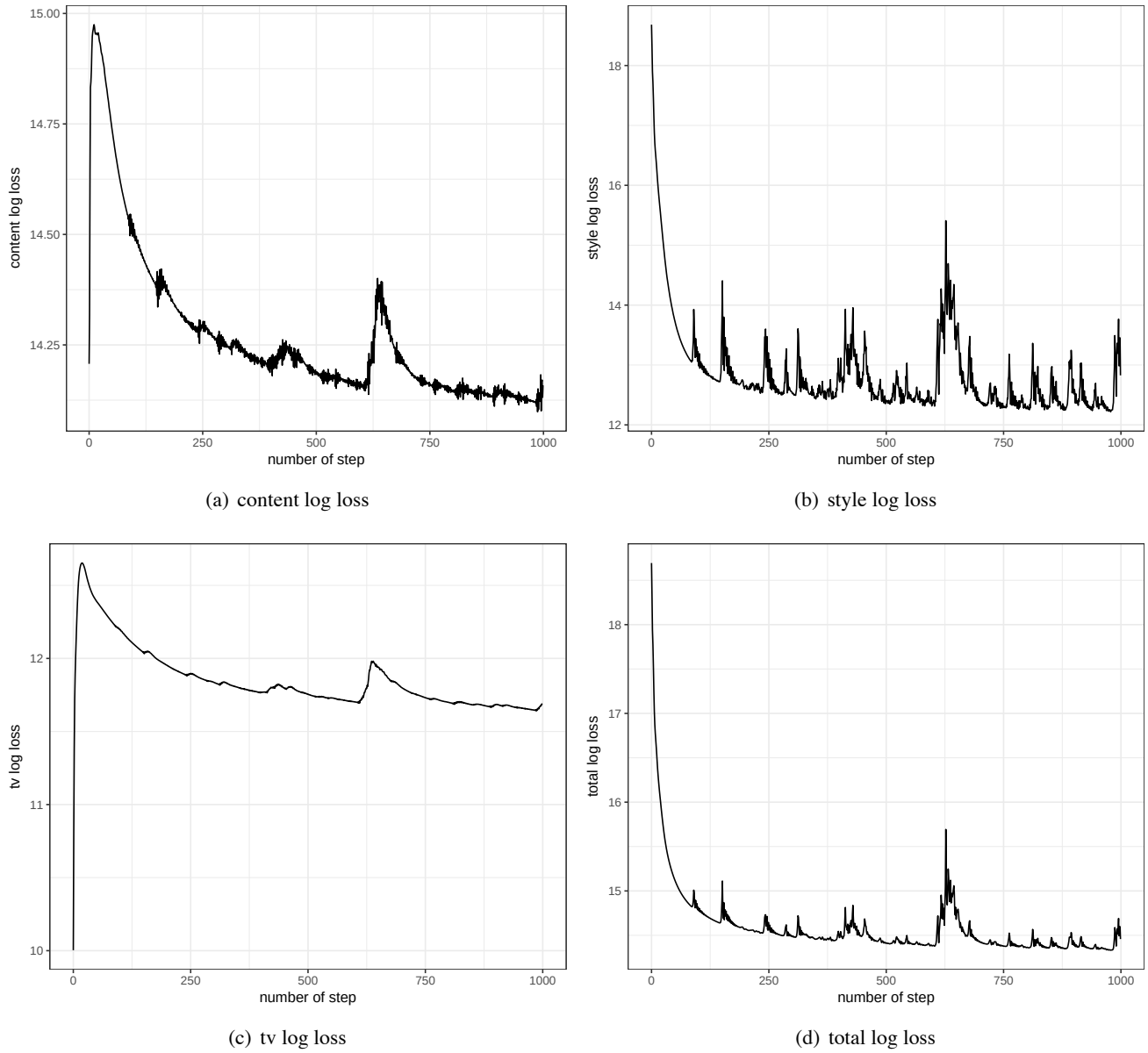
(b) style loss

(c) tv loss

(d) total loss

Figure 4–4: Model training loss plot

Since the absolute value of the loss is not informative enough and scaling of the plot prevents us from viewing the loss curve in details, Fig. 4–5 presents the logarithmic loss of each type. Notice that the loss data is un-smoothed and thus the plots have rugged shapes.



(a) content log loss

(b) style log loss

(c) tv log loss

(d) total log loss

Figure 4–5: Model training log loss plot

4.2 More examples

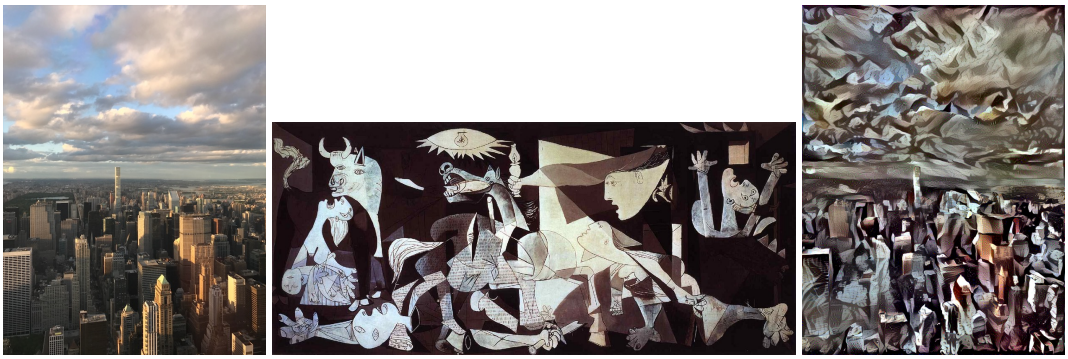
We also performed a few more experiments with other artistic styles. Among many artists who have a very strong characteristic and aesthetic style, we chose Pablo Picasso and Jackson Pollock as two examples. Furthermore, the works we chose (Guernica by Picasso[3] and Number 30 by Pollock[4]) both have a very strong and identifiable style.

In Fig. 4–6 and Fig. 4–7 we present the original photo, artistic work, and style tranferred output picture.

The procedure of these two examples are the same as of the Van Gogh example. Therefore, checkpoint output

[3] Pablo Picasso, Guernica (1937)

[4] Jackson Pollock, Autumn Rhythm (Number 30, 1950)



(a) New York photo (b) Pablo Picasso Guernica (c) Style tranferred output

Figure 4–6: Overlay Picasso on New York

(a) New York photo (b) Jackson Pollock Number 30 (c) Style tranferred output

Figure 4–7: Overlay Pollock on New York

images and loss function plot are not provided here.

The reader can easily replicate the examples with the Python code appended (see Appendix A and Appendix B).

5 Discussion

5.1 Tuning parameters optimization

In Sec. 3.2 we presented a pre-defined parameter list. For simplicity they are fixed in this work, but there is room for improvement in performance with parameter optimization.

5.1.1 Loss weighting

In the original work [3] the authors discussed the trade-off between content representation loss and style representation loss. Different ratios will lead to various focus on content versus style, and the user should be able to adjust the parameter to achieve a more suitable output result based on different needs.

In this work the ratio of content loss and style loss is set to 10^{-2} arbitrarily, and according to [3] this ratio exerts relatively higher emphasis on content feature extraction, therefore the output would resemble the raw input more.

The ratio choice would also be reflected on the difference convergence rate of loss function, as is shown in Fig. 4–4.

5.1.2 Convolutional layer selection

Different layers can be used for feature extraction. Usually convolutional layers and rectified linear units layers are used.

Apart from the choice of layer, number of layers can also be adjusted, for purpose of faster iteration or more elaborate representation.

5.1.3 Gradient descent parameters

The Adam optimizer takes four parameters, which are arbitrarily set in this project. The topic on the optimizer is beyond the scope of this work, and the reader can refer to [18] for more detailed specifications.

5.2 Run time optimization

All output results in this work are generated with a laptop with hardware specifications listed in Sec. 1.3. Average run time of a single batch of generation is roughly 36000 seconds, or 10 hours.

From the perspective of computing power escalation, common methods include using more CPUs, running with GPUs, implementing cloud computing services, or set up parallel computation mechanisms.

As to algorithm optimization, necessary parameter fixation and functionality elimination is required. In exchange for higher speed, the user would need to face with higher information loss (in both style representation and content representation) and more coarse output results.

There have been some works on fast style transfer, among which [19] is an excellent implementation. Perceptual loss and quality improvement in fast stylization are also discussed in [15, 20].

A Code Manuscript

Please refer to `https://github.com/tian-gao/AdvML-fall17-project` for detailed instructions. All source code are provided.

The following table shows the required directories and files to use the system.

	folder or file name	usage
folders	input	put content picture and style picture inside
	output	store output pictures
	logs	for Tensorboard logs
neural network	visual_geometry_group.py	pre-process the trained VGG network data
Tensorflow model	neural_network.py	feature extraction and model training with Tensorflow
system	style_transfer.py	main function to accept arguments
utility	utils.py	utility functions
	constants.py	VGG network layers and pre-defined parameters
	settings.py	file paths definition
	logger.py	formatted standard screen output
data	imagenet-vgg-verydeep-19.mat	VGG network data
other	requirements.txt	required Python libraries and version

Table A–1: Project directory

In order to carry out style transfer, do

`python style_transfer.py --content content.jpg --style style.jpg --output output.jpg`

to transfer "style" to "content" and get the "output".

Instructions to run the Python code and download the data are all explained in details on the GitHub repository.

B Python Code

B.1 style_transfer.py

```
import os
import sys
import time
import scipy.misc
from argparse import ArgumentParser
from utils import read_image, save_image

from logger import logger
from settings import PATH_INPUT_STYLE, PATH_INPUT_CONTENT, PATH_OUTPUT, TRAINED_NETWORK_DATA
from constants import (
    CONTENT_WEIGHT, STYLE_WEIGHT, TV_WEIGHT, POOLING,
    LEARNING_RATE, BETA1, BETA2, EPSILON, MAX_ITERATION
)
from visual_geometry_group import VGG
from neural_network import NeuralNetwork


def style_transfer(
        content_name, style_name, output_name, content_weight, style_weight, tv_weight,
        pooling, learning_rate, beta1, beta2, epsilon, max_iteration, check_point):
    time_start = time.time()

    # read images
    content = read_image(PATH_INPUT_CONTENT + content_name)
    style = read_image(PATH_INPUT_STYLE + style_name)
    style = scipy.misc.imresize(style, content.shape[1] / style.shape[1])

    # initialize objects
    vgg = VGG(TRAINED_NETWORK_DATA, pooling)
    nn = NeuralNetwork(content, style, vgg, content_weight, style_weight, tv_weight)

    # train model
    for k, output_image in nn.train_model(learning_rate, beta1, beta2, epsilon,
        max_iteration, check_point):
        name_list = output_name.split('.')
        image_name = PATH_OUTPUT + '.'.join(name_list[:-1]) + '_{}.{}'.format(str(k) if not
            k % check_point else 'final', name_list[-1])
        save_image(output_image, image_name)

    time_end = time.time()
    logger.info('Time elapsed: {} seconds'.format(round(time_end - time_start)))


def build_parser():
    parser = ArgumentParser()
    parser.add_argument('--content', dest='content', required=True,
                        help='Content image, e.g. "input.jpg"')
    parser.add_argument('--style', dest='style', required=True,
                        help='Style image, e.g. "style.jpg"')
    parser.add_argument('--output', dest='output', required=True,
                        help='Output image, e.g. "output.jpg"')
    return parser
```

```
51
52
53  if __name__ == '__main__':
54      parser = build_parser()
55      args = parser.parse_args()
56
57      # check if network data file exists
58      if not os.path.isfile(TRAINED_NETWORK_DATA):
59          logger.error('Cannot find pre-trained network data file!')
60          sys.exit()
61
62      style_transfer(
63          content_name=args.content,
64          style_name=args.style,
65          output_name=args.output,
66
67          content_weight=CONTENT_WEIGHT,
68          style_weight=STYLE_WEIGHT,
69          tv_weight=TV_WEIGHT,
70          pooling=POOLING,
71
72          learning_rate=LEARNING_RATE,
73          beta1=BETA1,
74          beta2=BETA2,
75          epsilon=EPSILON,
76          max_iteration=MAX_ITERATION,
77          check_point=MAX_ITERATION / 10
78      )
```

B.2 neural_network.py

```
import numpy as np
import tensorflow as tf
from functools import reduce
from operator import mul

from logger import logger
from constants import CONTENT_LAYERS, STYLE_LAYERS
from utils import process_image, unprocess_image


class NeuralNetwork(object):
    """NeuralNetwork provides an interface to formulate the Tensorflow neural network model
    and perform style transfer algorithm"""
    def __init__(self, content, style, vgg, content_weight, style_weight, tv_weight):
        logger.info('Initializing neural network......')
        self.content = content
        self.style = style
        self.vgg = vgg

        self.content_weight = content_weight
        self.style_weight = style_weight
        self.tv_weight = tv_weight

        self.content_shape, self.style_shape, self.content_layer_weights, self.
            style_layer_weights = self.get_parameters()
        self.content_features, self.style_features = self.get_features()

    def get_parameters(self):
        logger.info('Fetching images parameters......')
        content_shape = (1, ) + self.content.shape
        style_shape = (1, ) + self.style.shape

        # get content layer weights
        content_layer_weights = {}
        content_layer_weights['relu4_2'] = 1.0
        content_layer_weights['relu5_2'] = 0.0

        # get style layer weights
        style_layer_weights = {}
        for style_layer in STYLE_LAYERS:
            style_layer_weights[style_layer] = 1.0 / len(STYLE_LAYERS)

        return content_shape, style_shape, content_layer_weights, style_layer_weights

    def get_features(self):
        content_features = self._get_content_feature()
        style_features = self._get_style_feature()
        return content_features, style_features

    def _get_content_feature(self):
        logger.info('Fetching content features......')
        content_features = {}
        graph = tf.Graph()
        with graph.as_default(), graph.device('/cpu:0'), tf.Session() as session:
            content_image = tf.placeholder('float', shape=self.content_shape)
            content_net = self.vgg.load_net(content_image)
            content_pre = np.array([
```

```
                process_image(self.content, self.vgg.mean_pixel)])
            for content_layer in CONTENT_LAYERS:
                content_features[content_layer] = content_net[content_layer].eval(feed_dict
                    ={content_image: content_pre})

        return content_features

    def _get_style_feature(self):
        logger.info('Fetching style features......')
        style_features = {}
        graph = tf.Graph()
        with graph.as_default(), graph.device('/cpu:0'), tf.Session() as session:
            style_image = tf.placeholder('float', shape=self.style_shape)
            style_net = self.vgg.load_net(style_image)
            style_pre = np.array([
                process_image(self.style, self.vgg.mean_pixel)])
            for style_layer in STYLE_LAYERS:
                feature = style_net[style_layer].eval(feed_dict={style_image: style_pre})
                feature = np.reshape(feature, (-1, feature.shape[3]))
                gram = feature.T.dot(feature) / feature.size
                style_features[style_layer] = gram

        return style_features

    def train_model(self, learning_rate, beta1, beta2, epsilon, max_iteration, check_point):
        with tf.Graph().as_default():
            # initialize with random guess
            logger.info('Initializing tensorflow graph with random guess......')
            noise = np.random.normal(size=self.content_shape, scale=np.std(self.content) *
                0.1)
            initial_guess = tf.random_normal(self.content_shape) * 0.256
            input_image = tf.Variable(initial_guess)
            parsed_net = self.vgg.load_net(input_image)

            # calculate loss
            content_loss = self._calculate_content_loss(parsed_net)
            style_loss = self._calculate_style_loss(parsed_net)
            tv_loss = self._calculate_tv_loss(input_image)
            loss = content_loss + style_loss + tv_loss

            # summary statistics
            tf.summary.scalar('content_loss', content_loss)
            tf.summary.scalar('style_loss', style_loss)
            tf.summary.scalar('tv_loss', tv_loss)
            tf.summary.scalar('total_loss', loss)
            summary_loss = tf.summary.merge_all()

            # initialize optimization
            train_step = tf.train.AdamOptimizer(learning_rate, beta1, beta2, epsilon).
                minimize(loss)

            with tf.Session() as session:
                summary_writer = tf.summary.FileWriter('logs/neural_network', session.graph)
                logger.info('Saving graph......')

                session.run(tf.global_variables_initializer())
                logger.info('Initializing optimization......')
                logger.info('Current total loss: {}'.format(loss.eval()))

```

```
                for k in range(max_iteration):
                    logger.info('Iteration {} total loss {}'.format(str(k+1), loss.eval()))
                    train_step.run()
                    summary = session.run(summary_loss)
                    summary_writer.add_summary(summary, k)

                    # save intermediate images at checkpoints
                    if (check_point and (not k % check_point)) or k == max_iteration - 1:
                        output_temp = input_image.eval()
                        output_image = unprocess_image(output_temp.reshape(self.
                            content_shape[1:]), self.vgg.mean_pixel)
                        yield k, output_image

    def _calculate_content_loss(self, parsed_net):
        logger.info('Calculating content loss......')
        losses = []
        for content_layer in CONTENT_LAYERS:
            losses += [
                self.content_layer_weights[content_layer] * self.content_weight * (
                    2 * tf.nn.l2_loss(
                        parsed_net[content_layer] - self.content_features[content_layer]
                    ) / self.content_features[content_layer].size)]
        return reduce(tf.add, losses)

    def _calculate_style_loss(self, parsed_net):
        logger.info('Calculating style loss......')
        losses = []
        for style_layer in STYLE_LAYERS:
            layer = parsed_net[style_layer]
            _, height, width, number = map(lambda x: x.value, layer.get_shape())
            size = height * width * number
            feats = tf.reshape(layer, (-1, number))
            gram = tf.matmul(tf.transpose(feats), feats) / size
            style_gram = self.style_features[style_layer]
            losses += [
                self.style_layer_weights[style_layer] * 2 * tf.nn.l2_loss(gram - style_gram)
                    / style_gram.size]
        return self.style_weight * reduce(tf.add, losses)

    def _calculate_tv_loss(self, image):
        # total variation denoising
        logger.info('Calculating total variation loss......')
        tv_y_size = self._get_tensor_size(image[:, 1:, :, :])
        tv_x_size = self._get_tensor_size(image[:, :, 1:, :])
        tv_loss = self.tv_weight * 2 * ((
            tf.nn.l2_loss(image[:, 1:, :, :] - image[:, :self.content_shape[1]-1, :, :]) /
                tv_y_size) + (
            tf.nn.l2_loss(image[:, :, 1:, :] - image[:, :, :self.content_shape[2]-1, :]) /
                tv_x_size))
        return tv_loss

    def _get_tensor_size(self, tensor):
        return reduce(mul, (d.value for d in tensor.get_shape()), 1)
```

B.3 visual_geometry_group.py

```python
import numpy as np
import scipy.io
import tensorflow as tf

from logger import logger
from constants import VGG19_LAYERS


class VGG(object):
    """VGG provides an interface to extract parameter from pre-trained neural network
    and formulate Tensorflow layers"""
    def __init__(self, trained, pooling):
        logger.info('Loading pre-trained network data......')
        self.network = scipy.io.loadmat(trained)
        self.layers, self.mean_pixel = self.init_net()
        self.pooling = pooling

    def init_net(self):
        mean_mat = self.network['normalization'][0][0][0]  # shape: (224, 224, 3)
        mean_pixel = np.mean(mean_mat, axis=(0, 1))  # length: 3
        layers = self.network['layers'].reshape(-1)  # length: 43
        return layers, mean_pixel

    def load_net(self, input_image):
        # construct layers using parameters
        logger.info('Parsing layers......')
        parsed_net = {}
        current_image = input_image

        for layer_name, input_layer in zip(VGG19_LAYERS, self.layers):
            layer_kind = layer_name[:4]

            if layer_kind == 'conv':
                current_image = self._get_conv_layer(current_image, input_layer)
            elif layer_kind == 'relu':
                current_image = self._get_relu_layer(current_image)
            elif layer_kind == 'pool':
                current_image = self._get_pool_layer(current_image)
            parsed_net[layer_name] = current_image

        assert len(parsed_net) == len(VGG19_LAYERS)
        return parsed_net

    def _get_conv_layer(self, input_image, input_layer):
        # get kernel and bias
        # matconvnet: weights are [width, height, in_channels, out_channels]
        # tensorflow: weights are [height, width, in_channels, out_channels]
        kernels, bias = input_layer[0][0][0][0]
        kernels = np.transpose(kernels, (1, 0, 2, 3))
        bias = bias.reshape(-1)

        # formulate conv layer
        conv = tf.nn.conv2d(input_image, tf.constant(kernels), strides=(1, 1, 1, 1), padding
            ='SAME')
        layer = tf.nn.bias_add(conv, bias)
        return layer

```

```
57      def _get_relu_layer(self, input_image):
58          return tf.nn.relu(input_image)
59
60      def _get_pool_layer(self, input_image):
61          if self.pooling == 'avg':
62              layer = tf.nn.avg_pool(input_image, ksize=(1, 2, 2, 1), strides=(1, 2, 2, 1),
                    padding='SAME')
63          elif self.pooling == 'max':
64              layer = tf.nn.max_pool(input_image, ksize=(1, 2, 2, 1), strides=(1, 2, 2, 1),
                    padding='SAME')
65          return layer
```

References

[1] Wikipedia. Prisma (app) — wikipedia, the free encyclopedia, 2017.

[2] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. A Neural Algorithm of Artistic Style. *arXiv.org*, August 2015.

[3] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image Style Transfer Using Convolutional Neural Networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR*, pages 2414–2423. IEEE, 2016.

[4] Wikipedia. Deepart — wikipedia, the free encyclopedia, 2017.

[5] Anish Athalye. Neural style. `https://github.com/anishathalye/neural-style`, 2015.

[6] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully Convolutional Networks for Semantic Segmentation. *arXiv.org*, November 2014.

[7] Leon Gatys, Alexander S Ecker, and Matthias Bethge. Texture synthesis using convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 262–270, 2015.

[8] Dmitry Ulyanov, Vadim Lebedev, Andrea Vedaldi, and Victor Lempitsky. Texture Networks: Feed-forward Synthesis of Textures and Stylized Images. *arXiv.org*, March 2016.

[9] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252, December 2015.

[10] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1701–1708, 2014.

[11] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[13] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv.org*, September 2014.

[14] University of Oxford Department of Engineering Science. Visual geometry group: Very deep convolutional networks for large-scale visual recognition, 2014.

[15] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual Losses for Real-Time Style Transfer and Super-Resolution. In *Computer Vision – ECCV 2016*, pages 694–711. Springer International Publishing, Cham, September 2016.

[16] Hussein A Aly and Eric Dubois. Image up-sampling using total-variation regularization with a new observation model. *IEEE Transactions on Image Processing*, 14(10):1647–1659, 2005.

[17] Haichao Zhang, Jianchao Yang, Yanning Zhang, and Thomas S Huang. Non-local kernel regression for image and video restoration. In *European Conference on Computer Vision*, pages 566–579. Springer, 2010.

[18] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[19] Logan Engstrom. Fast style transfer. `https://github.com/lengstrom/fast-style-transfer/`, 2016.

[20] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance Normalization: The Missing Ingredient for Fast Stylization. *arXiv.org*, July 2016.