Artistic Style Transfer Implementation with Tensorflow

GR5242 Advanced Machine Learning Final Project

Tian Gao
tg2585
tian.gao@columbia.edu

GR5242 Advanced Machine Learning Final Project
October 26, 2017

Contents

List of Figures

List of Tables

1 Introduction

1.1 Project background

1.2 Related works

1.3 Development environment and hardware information

2 Mathematical Foundation

2.1 Covolutional neural network

2.2 VGG-network

2.3 Loss function

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.

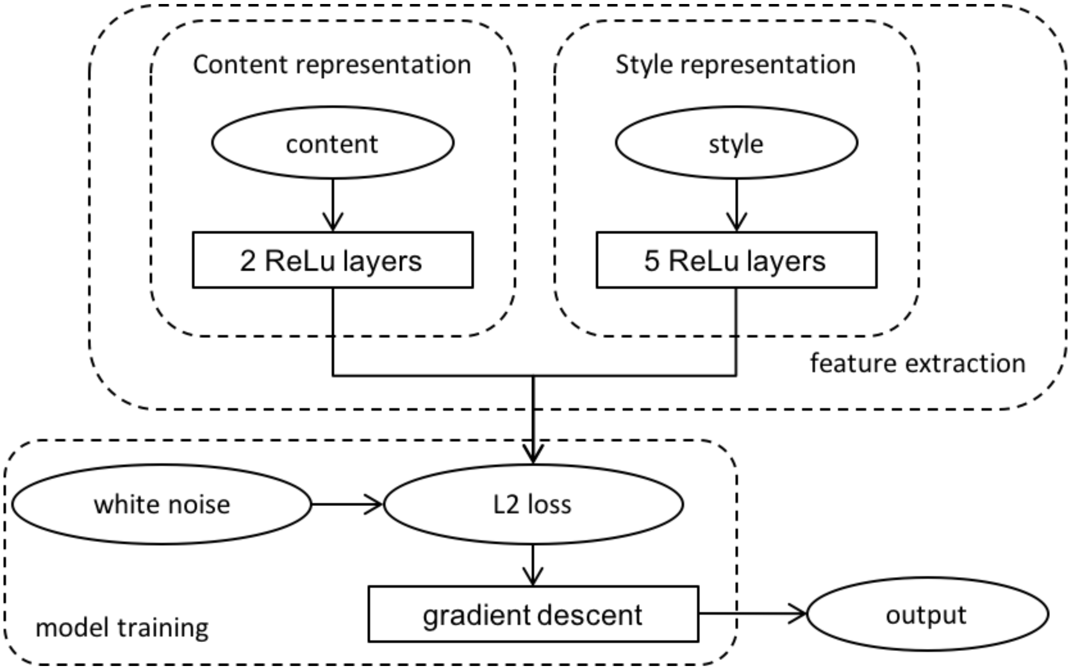

Figure 3–1: Tensorflow implementation system overview

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 [1].

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 [2], 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 [1], or the reference [3] 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 [1] 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

4.2 More examples

5 Discussion

5.1 Tuning parameters optimization

5.2 Running time optimization

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 output	put content picture and style picture inside store output pictures
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 constants.py settings.py logger.py	utility functions VGG network layers and pre-defined parameters file paths definition formatted standard screen output
data	imagenet-vgg-verydeep-19.mat	VGG network data

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

```python
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


if __name__ == '__main__':
    parser = build_parser()
    args = parser.parse_args()

    # check if network data file exists
    if not os.path.isfile(TRAINED_NETWORK_DATA):
        logger.error('Cannot find pre-trained network data file!')
        sys.exit()

    style_transfer(
        content_name=args.content,
        style_name=args.style,
        output_name=args.output,

        content_weight=CONTENT_WEIGHT,
        style_weight=STYLE_WEIGHT,
        tv_weight=TV_WEIGHT,
        pooling=POOLING,

        learning_rate=LEARNING_RATE,
        beta1=BETA1,
        beta2=BETA2,
        epsilon=EPSILON,
        max_iteration=MAX_ITERATION,
        check_point=MAX_ITERATION / 10
    )
```

B.2 neural_network.py

```python
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

    def _get_relu_layer(self, input_image):
        return tf.nn.relu(input_image)

    def _get_pool_layer(self, input_image):
        if self.pooling == 'avg':
            layer = tf.nn.avg_pool(input_image, ksize=(1, 2, 2, 1), strides=(1, 2, 2, 1), padding='SAME')
        elif self.pooling == 'max':
            layer = tf.nn.max_pool(input_image, ksize=(1, 2, 2, 1), strides=(1, 2, 2, 1), padding='SAME')
        return layer
```

References

[1] 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.

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

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