画风迁移详细设计

1.引言

1.1 编写目的

在前一阶段（概要设计说明书）中，已解决了实现该系统需求的程序模块设计问题。包括如何把系统划分成若干个模块、决定各个模块之间的接口、模块之间传递的信息，以及数据结构、模块结构的设计等。在以下的详细设计报告中将对在本阶段中对系统所做的所有详细设计进行说明。

在本阶段中，确定应该如何具体地实现所要求的系统，从而在编码阶段可以把这个描述直接翻译成用具体的程序语言书写的程序。此需求说明书详细陈述了“实验设备管理系统”的所提供各项功能。其中包括用户的功能性需求以及非功能性需求，为用户提供完整且较详尽的系统功能运作蓝图。同时为设计人员提供一个完整的、可靠的设计约束，以便高质量地设计、编写代码，完成项目预期目标。还给开发人员提供了参考。

此需求说明书的预期读者为项目经理、设计人员、开发人员、用户等。

1.2 背景

A．待开发系统的名称：画风迁移

B．本项目任务的提出者：李源钊、黄京津

C．本项目任务的开发者：张忠宇、张拓、李卓航

D．本项目的用户：当今对照片风格转化有需求的青年群体或小规模公司

1.3 定义

1.4 参考资料

概要设计说明书	《画风迁移》软件开发小组	
需求规格说明书	《画风迁移》软件开发小组	
《软件工程》	张秋余、张聚礼等	西安电子科技大学出版社

2.系统的结构

2.1 需求概述

A．功能说明：通过该系统用户能够通过输入原图，选择风格图，然后系统

进行风格转换后，用户获得相应的目标图。

B．性能需求：

（1）目标图片的内容需要与原图高度一致。

（2）目标图片需要与所选择的风格图画风一致。

（3)计算时间要尽可能的短(可以根据比较流行的风格图预先训练模型)。

（4）系统的适用性要强，可在大多数计算机上运行。

2.2 系统结构图

在以下给出的画风迁移的系统结构图中，可以清楚的看出各个模块之间层次关系，以及各个模块之间需要传递的数据流，接下来会给出各个模块的详细设计方案。

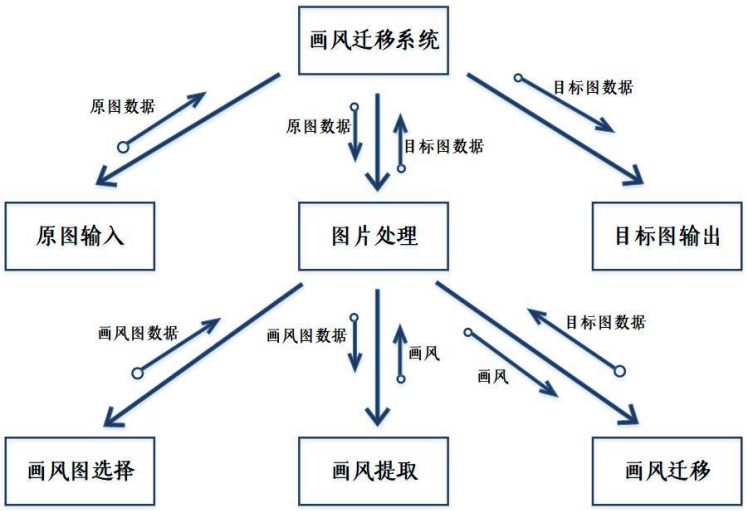


图 1 画风迁移系统结构图

我们开发的画风迁移系统主要分为三大模块：用户原图输入，照片处理以及风格转化，目标图输出。

2.3 关键程序的设计

（1）程序（load_vgg）设计说明：

功能：该程序的主要是用来调用提前训练好的 VGG 网络的数据，用来对输入原图进行处理，从而得到目标图。

性能：由于只是简单地调用，所以性能很好。

输入与输出：由于只是简单地调用，所以输入输出暂无，也可以理解为输入为风格图片，输出为 VGG19 网络的参数信息。

算法逻辑：在该程序里，主要定义了几个重要的功能，包括定义网络的输入、定义网络的激活函数、定义 pooling 层以及加载 pre_trained 的数据。

接口：处理主程序与画风提取模块连接

代码实现：

```
class VGG(object):
    def __init__(self, input_img):
        # 下载文件
        utils.download(VGG_DOWNLOAD_LINK, VGG_FILENAME,
EXPECTED_BYTES)
        # 加载文件
        self.vgg_layers = scipy.io.loadmat(VGG_FILENAME)["layers"]
        self.input_img = input_img
        # VGG 在处理图像时候会将图片进行 mean-center，所以我们首先要计算 RGB
三个 channel 上的 mean
        self.mean_pixels = np.array([123.68, 116.779, 103.939]).reshape((1, 1, 1, 3))

    def _weights(self, layer_idx, expected_layer_name):
        """
        获取指定 layer 层的 pre-trained 权重

        :param layer_idx: VGG 中的 layer id
        :param expected_layer_name: 当前 layer 命名
        :return: pre-trained 权重 W 和 b
        """
        W = self.vgg_layers[0][layer_idx][0][0][2][0][0]
        b = self.vgg_layers[0][layer_idx][0][0][2][0][1]
        # 当前层的名称
        layer_name = self.vgg_layers[0][layer_idx][0][0][0][0]
        assert layer_name == expected_layer_name, print("Layer name error!")

        return W, b.reshape(b.size)

    def conv2d_relu(self, prev_layer, layer_idx, layer_name):
        """
        采用 relu 作为激活函数的卷积层

        :param prev_layer: 前一层网络
        :param layer_idx: VGG 中的 layer id
        :param layer_name: 当前 layer 命名
        """
        with tf.variable_scope(layer_name):
            # 获取当前权重（numpy 格式）
```

```
                W, b = self._weights(layer_idx, layer_name)
                # 将权重转化为tensor（由于我们不需要重新训练VGG 的权重，因此初始化为常数）
                W = tf.constant(W, name="weights")
                b = tf.constant(b, name="bias")
                # 卷积操作
                conv2d = tf.nn.conv2d(input=prev_layer,
                                      filter=W,
                                      strides=[1, 1, 1, 1],
                                      padding="SAME")
                # 激活
                out = tf.nn.relu(conv2d + b)
            setattr(self, layer_name, out)

        def avgpool(self, prev_layer, layer_name):
            """
            average pooling 层（这里参考了原论文中提到了avg-pooling 比max-pooling 效果好，所以采用avg-pooling）

            :param prev_layer: 前一层网络（卷积层）
            :param layer_name: 当前layer 命名
            """
            with tf.variable_scope(layer_name):
                # average pooling
                out = tf.nn.avg_pool(value=prev_layer,
                                     ksize=[1, 2, 2, 1],
                                     strides=[1, 2, 2, 1],
                                     padding="SAME")

            setattr(self, layer_name, out)

        def load(self):
            """
            加载pre-trained 的数据
            """
            self.conv2d_relu(self.input_img, 0, "conv1_1")
            self.conv2d_relu(self.conv1_1, 2, "conv1_2")
            self.avgpool(self.conv1_2, "avgpool1")
            self.conv2d_relu(self.avgpool1, 5, "conv2_1")
            self.conv2d_relu(self.conv2_1, 7, "conv2_2")
            self.avgpool(self.conv2_2, "avgpool2")
            self.conv2d_relu(self.avgpool2, 10, "conv3_1")
            self.conv2d_relu(self.conv3_1, 12, "conv3_2")
            self.conv2d_relu(self.conv3_2, 14, "conv3_3")
```

```
self.conv2d_relu(self.conv3_3, 16, "conv3_4")
self.avgpool(self.conv3_4, "avgpool3")
self.conv2d_relu(self.avgpool3, 19, "conv4_1")
self.conv2d_relu(self.conv4_1, 21, "conv4_2")
self.conv2d_relu(self.conv4_2, 23, "conv4_3")
self.conv2d_relu(self.conv4_3, 25, "conv4_4")
self.avgpool(self.conv4_4, "avgpool4")
self.conv2d_relu(self.avgpool4, 28, "conv5_1")
self.conv2d_relu(self.conv5_1, 30, "conv5_2")
self.conv2d_relu(self.conv5_2, 32, "conv5_3")
self.conv2d_relu(self.conv5_3, 34, "conv5_4")
self.avgpool(self.conv5_4, "avgpool5")
```

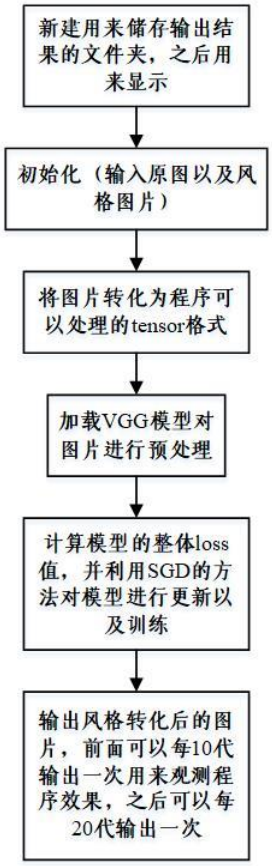
（2）程序（style_transfer）设计说明

功能：将原图与风格图经过程序的训练输出用户想要得到的相应风格的目标图。

性能：运行时间为 3.2s，运行速度可以接受，后续可以再次优化，比如对网络进行相应的改进或者对图片进行压缩处理。

输入与输出：输入为用户原图与风格图；输出为风格转化完成的目标图。

算法逻辑：



接口：处理主程序与输入图片模块连接、主程序与画风提取模块链接、主程序与目标图输出模块链接。

代码实现：本段代码实现过长，放于附录。

3.结束语

经过之前的总体设计，已经基本实现了整个系统的全部功能，之后整个性能的测试与完善将在软件测试的时候进行详细说明，本详细设计说明书，主要面向开发人员，使开发人员有一个完整清晰的逻辑，有利于整个系统的实现。

4.附录

1.主要程序 style_transfer 代码：

```
import os
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "2"
import numpy as np
import tensorflow as tf
import load_vgg
import utils

def setup():
    """
    新建存储模型的文件夹 checkpoints 和存储合成图片结果的文件夹 outputs
    """
    utils.safe_mkdir("checkpoints")
    utils.safe_mkdir("outputs")

class StyleTransfer(object):
    def __init__(self, content_img, style_img, img_width, img_height):
        """
        初始化
        :param content_img: 待转换风格的图片（保留内容的图片）
        :param style_img: 风格图片（保留风格的图片）
        :param img_width: 图片的 width
        :param img_height: 图片的 height
        """
        # 获取基本信息
        self.content_name = str(content_img.split("/")[-1].split(".")[0])
        self.style_name = str(style_img.split("/")[-1].split(".")[0])
        self.img_width = img_width
        self.img_height = img_height
        # 规范化图片的像素尺寸
        self.content_img = utils.get_resized_image(content_img, img_width, img_height)
```

```
        self.style_img = utils.get_resized_image(style_img, img_width, img_height)
        self.initial_img = utils.generate_noise_image(self.content_img, img_width,
img_height)

        # 定义提取特征的层
        self.content_layer = "conv4_2"
        self.style_layers = ["conv1_1", "conv2_1", "conv3_1", "conv4_1", "conv5_1"]

        # 定义 content loss 和 style loss 的权重
        self.content_w = 0.001
        self.style_w = 1

        # 不同 style layers 的权重，层数越深权重越大
        self.style_layer_w = [0.5, 1.0, 1.5, 3.0, 4.0]

        # global step 和学习率
        self.gstep = tf.Variable(0, dtype=tf.int32, trainable=False, name="global_step")
# global step
        self.lr = 2.0

        utils.safe_mkdir("outputs/%s_%s" % (self.content_name, self.style_name))

    def create_input(self):
        """
        初始化图片 tensor
        """
        with tf.variable_scope("input"):
            self.input_img = tf.get_variable("in_img",
                                             shape=([1, self.img_height,
self.img_width, 3]),
                                             dtype=tf.float32,
                                             initializer=tf.zeros_initializer())

    def load_vgg(self):
        """
        加载 vgg 模型并对图片进行预处理
        """
        self.vgg = load_vgg.VGG(self.input_img)
        self.vgg.load()
        # mean-center
        self.content_img -= self.vgg.mean_pixels
        self.style_img -= self.vgg.mean_pixels

    def _content_loss(self, P, F):
```

```
        """
        计算 content loss

        :param P: 内容图像的 feature map
        :param F: 合成图片的 feature map
        """
        self.content_loss = tf.reduce_sum(tf.square(F - P)) / (4.0 * P.size)

    def _gram_matrix(self, F, N, M):
        """
        构造 F 的 Gram Matrix（格雷姆矩阵），F 为 feature map，shape=(widths, heights, channels)

        :param F: feature map
        :param N: feature map 的第三维度
        :param M: feature map 的第一维 乘 第二维
        :return: F 的 Gram Matrix
        """
        F = tf.reshape(F, (M, N))

        return tf.matmul(tf.transpose(F), F)

    def _single_style_loss(self, a, g):
        """
        计算单层 style loss

        :param a: 当前 layer 风格图片的 feature map
        :param g: 当前 layer 生成图片的 feature map
        :return: style loss
        """
        N = a.shape[3]
        M = a.shape[1] * a.shape[2]

        # 生成 feature map 的 Gram Matrix
        A = self._gram_matrix(a, N, M)
        G = self._gram_matrix(g, N, M)

        return tf.reduce_sum(tf.square(G - A)) / ((2 * N * M) ** 2)

    def _style_loss(self, A):
        """
        计算总的 style loss

        :param A: 风格图片的所有 feature map
```

```
            """
            # 层数（我们用了 conv1_1, conv2_1, conv3_1, conv4_1, conv5_1）
            n_layers = len(A)
            # 计算 loss
            E = [self._single_style_loss(A[i], getattr(self.vgg, self.style_layers[i]))
                 for i in range(n_layers)]
            # 加权求和
            self.style_loss = sum(self.style_layer_w[i] * E[i] for i in range(n_layers))

        def losses(self):
            """
            模型总体 loss
            """
            with tf.variable_scope("losses"):
                # contents loss
                with tf.Session() as sess:
                    sess.run(self.input_img.assign(self.content_img))
                    gen_img_content = getattr(self.vgg, self.content_layer)
                    content_img_content = sess.run(gen_img_content)
                self._content_loss(content_img_content, gen_img_content)

                # style loss
                with tf.Session() as sess:
                    sess.run(self.input_img.assign(self.style_img))
                    style_layers = sess.run([getattr(self.vgg, layer) for layer in
self.style_layers])
                self._style_loss(style_layers)

                # 加权求得最终的 loss
                self.total_loss = self.content_w * self.content_loss + self.style_w *
self.style_loss

        def optimize(self):
            self.optimizer = tf.train.AdamOptimizer(self.lr).minimize(self.total_loss,
global_step=self.gstep)

        def create_summary(self):
            with tf.name_scope("summary"):
                tf.summary.scalar("contents loss", self.content_loss)
                tf.summary.scalar("style loss", self.style_loss)
                tf.summary.scalar("total loss", self.total_loss)
                self.summary_op = tf.summary.merge_all()

        def build(self):
```

```
        self.create_input()
        self.load_vgg()
        self.losses()
        self.optimize()
        self.create_summary()

    def train(self, epoches=300):
        skip_step = 1
        with tf.Session() as sess:
            sess.run(tf.global_variables_initializer())
            writer = tf.summary.FileWriter("graphs/style_transfer", sess.graph)

            sess.run(self.input_img.assign(self.initial_img))

            saver = tf.train.Saver()
            ckpt =
tf.train.get_checkpoint_state(os.path.dirname("checkpoints/%s_%s_style_transfer/checkpoint" %

(self.content_name, self.style_name)))
            if ckpt and ckpt.model_checkpoint_path:
                print("You have pre-trained model, if you do not want to use this, please
delete the existing one.")
                saver.restore(sess, ckpt.model_checkpoint_path)

            initial_step = self.gstep.eval()

            for epoch in range(initial_step, epoches):
                # 前面几轮每隔 10 个 epoch 生成一张图片
                if epoch >= 5 and epoch < 20:
                    skip_step = 10
                # 后面每隔 20 个 epoch 生成一张图片
                elif epoch >= 20:
                    skip_step = 20

                sess.run(self.optimizer)
                if (epoch + 1) % skip_step == 0:
                    gen_image, total_loss, summary = sess.run([self.input_img,
                                                               self.total_loss,

self.summary_op])
                    # 对生成的图片逆向 mean-center，即在每个 channel 上加上 mean
                    gen_image = gen_image + self.vgg.mean_pixels
                    writer.add_summary(summary, global_step=epoch)
```

```
                print("Step {}\n      Sum: {:5.1f}".format(epoch + 1,
np.sum(gen_image)))
                print("      Loss: {:5.1f}".format(total_loss))

                filename = "outputs/%s_%s/epoch_%d.png" % (self.content_name,
self.style_name, epoch)
                utils.save_image(filename, gen_image)

                # 存储模型
                if (epoch + 1) % 20 == 0:
                    saver.save(sess,

"checkpoints/%s_%s_style_transfer/style_transfer" %
                                                (self.content_name, self.style_name), epoch)

    if __name__ == "__main__":
        setup()
        # 指定图片
        content_img = "contents/sky.jpg"
        style_img = "styles/starry_night.jpg"
        # 指定像素尺寸
        img_width = 400
        img_height = 300
        # style transfer
        style_transfer = StyleTransfer(content_img, style_img, img_width, img_height)
        style_transfer.build()
        style_transfer.train(300)
```