**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ**

**«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»**

###### Центр непрерывного образования

###### Факультета компьютерных наук

**АТТЕСТАЦИОННАЯ РАБОТА**

Style transfering - нейронный перенос стиля

Название темы

|  |  |
| --- | --- |
|  | Выполнил: |
|  | Глаголева Анна |
|  | Ф.И.О. |
|  | Руководитель: |
|  | Слинько Игорь  степень, звание, должность Ф.И.О. |

Москва 2019

**1 Введение**

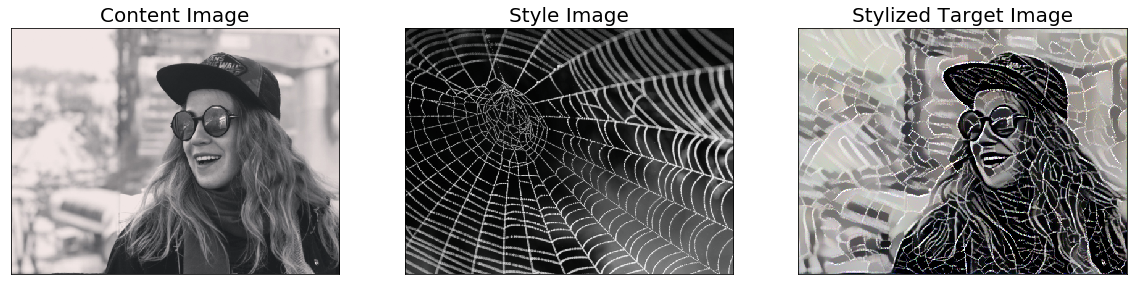
На сегодняшний день современное общество можно назвать пожирателями контента, и модификация фотографий с целью их улучшения в этом мире занимает далеко не самое последнее место. Такие манипуляции над изображением как размытие, ретушь, выделение деталей или наложение фильтров просто решаются попиксельными и морфологическими операциями, то задача переноса стиля, позволяющее сделать из фотографии настоящие картины на подобии произведения искусства требует более сложного метода.

Нейронная передача стиля – это одна из важных разработок в области изменения изображений с использованием глубокого обучения. Эта идея была реализована Леоном Гатисом со своими коллегами в 2015 году.

В основе реализации передачи стиля лежит та же идея, которая занимает центральное положение во всех алгоритмах глубокого обучения: вы задаете функцию потерь, чтобы определить цель для достижения, затем минимизируете ее. В данном случае наша цель – это сохранить содержимое исходного изображения и передать стиль другого изображения.

**Постановка задачи**

Задача переноса стиля изображений состоит в следующем: у нас есть два изображения – контент (Content или *xcontent*) и стиль (Style или *xstyle*). Алгоритм создает третье изображение, где содержание первой картинки (Content) остается тем же, но стиль изображения придается от второго изображения (Style). Под стилем обычно понимают цветовое наполнение, текстуры и т.д. Как правило, в качестве таких изображений выступают работы известных художников. Пример показан на рисунке 1.



На практике этот алгоритм используется очень часто в мобильных приложениях и дизайнерами, для стилизации изображения, превращения их в «картин» и других художественных задач.

В ходе работы будут проходить эксперименты по созданию новых графических текстур для редакторов изображений.

**2. Обзор литературы и существующих методов**

Создание алгоритма с нуля потребовало бы много времени и вычислительных ресурсов, что не всегда доступно для всех.

Вместо этого мы будем загружать веса существующей сети для реализации переноса стиля.

Процесс использования уже обученной ранее нейронной сети для других целей называется **transfer learning**.

Теперь рассмотрим реализацию оригинального алгоритма нейронной передачи стиля 2015 года с применением библиотеки keras.

Нейронную передачу стиля можно реализовать с использованием любой обученной сверточной сети. Здесь мы будем использовать сеть VGG16

Вот как выглядит весь процесс в общих чертах:

1. Настройка сети, которая вычисляет активации слоя VGG16 одновременно для изображения-образца, целевого и сгенерированного изображений.

2. Активации, вычисленные по всем трем изображениям, используются для определения общей функции потерь, описанной выше, которая будет минимизироваться для достижения эффекта передачи стиля.

3. Настройка процедуры градиентного восхождения для минимизации этой функции потерь.

Сначала определим пути к изображению-образцу и целевому изображению. Чтобы гарантировать совместимость размеров обрабатываемых изображений (сильно различающиеся размеры затрудняют передачу стиля), приведем их к общей высоте в 400 пикселов.

**2.1 Определение начальных переменных**

from keras.preprocessing.image import load\_img, img\_to\_array

target\_image\_path = 'img/portrait.jpg' *# Путь к изображению, которое будет трансформироваться*

style\_reference\_image\_path = 'img/transfer\_style\_reference.jpg'*# Путь к изображению с образцом стиля*

width, height = load\_img(target\_image\_path).size

img\_height = 400

img\_width = int(width \* img\_height / height) # *Размеры генерируемого изображения*

Нам понадобится несколько вспомогательных функций для загрузки, а также для предварительной и заключительной обработки изображений перед передачей изображений в сеть VGG16 и после вывода их из сети.

**2.2 Вспомогательные функции**

import numpy as np

from keras.applications import vgg16

def preprocess\_image(image\_path):

img = load\_img(image\_path, target\_size=(img\_height, img\_width))

img = img\_to\_array(img)

img = np.expand\_dims(img, axis=0)

img = vgg16.preprocess\_input(img)

return img

def deprocess\_image(x): # *Нулевое центрирование путем удаления среднего значения пиксела из ImageNet. Это отменяет преобразование, выполненное vgg16.preprocess\_input*

x[:, :, 0] += 103.939

x[:, :, 1] += 116.779

x[:, :, 2] += 123.68 .

x = x[:, :, ::-1] # *Конвертирует изображения из BGR в RGB. Также является частью обратного порядка vgg16.preprocess\_input*

x = np.clip(x, 0, 255).astype('uint8')

return x

Настроим сеть VGG16. Она принимает на входе пакет из трех изображений: изображение с образцом стиля, целевое изображение и заготовка, куда будет помещено сгенерированное изображение. Заготовка — это символический тензор, значениями которого являются внешние массивы Numpy. Изображение-образец и целевое изображение являются статическими и поэтому определяются как K.constant, тогда как значения в заготовке генерируемого изображения будут изменяться с течением времени.

**2.3 Загрузка предварительно обученной сети VGG16 и применение ее к трем изображениям**

from keras import backend as K

target\_image = K.constant(preprocess\_image(target\_image\_path))

style\_reference\_image = K.constant(preprocess\_image

(style\_reference\_image\_path))

combination\_image = K.placeholder((1, img\_height, img\_width, 3)) # *Заготовка, куда будет помещено сгенерированное изображение*

input\_tensor = K.concatenate([target\_image,

style\_reference\_image,

combination\_image], axis=0)#*объединение трех изображений в один пакет*

model = vgg16.VGG16(input\_tensor=input\_tensor,

weights='imagenet',

include\_top=False)

print('Model loaded.') # *Конструирование сети VGG16 с пакетом из трех изображений на входе. В модель будут загружены веса, полученные в результате обучения на наборе ImageNet*

Теперь определим функцию потерь содержимого, которая позволит гарантировать сходство представлений целевого и сгенерированного изображений в верхнем слое сети VGG16.

**2.4 Функция потерь содержимого**

def content\_loss(base, combination):

return K.sum(K.square(combination - base))

Далее приводится функция потерь стиля. Она использует вспомогательную функцию для вычисления матрицы Грама из входной матрицы: матрицы корреляций, найденных в матрице оригинальных признаков.

**2.5 Функция потерь стиля**

def gram\_matrix(x):

features = K.batch\_flatten(K.permute\_dimensions(x, (2, 0, 1)))

gram = K.dot(features, K.transpose(features))

return gram

def style\_loss(style, combination):

S = gram\_matrix(style)

C = gram\_matrix(combination)

channels = 3

size = img\_height \* img\_width

return K.sum(K.square(S - C)) / (4. \* (channels \*\* 2) \* (size \*\* 2))

К этим двум компонентам потерь добавляется третий: функция общей потери вариации (total variation loss), которая оперирует пикселами генерируемого изображения. Она стимулирует пространственную целостность генерируемого изображения, что позволяет избежать появления мозаичного эффекта. Ее можно интерпретировать как регуляризацию потерь.

**2.6 Функция общей потери вариации**

def total\_variation\_loss(x):

a = K.square(

x[:, :img\_height - 1, :img\_width - 1, :] -

x[:, 1:, :img\_width - 1, :])

b = K.square(

x[:, :img\_height - 1, :img\_width - 1, :] -

x[:, :img\_height - 1, 1:, :])

return K.sum(K.pow(a + b, 1.25))

Функция потерь, которую мы должны минимизировать, возвращает среднее взвешенное этих трех компонентов. Для вычисления потери содержимого используется только один верхний слой block5\_conv2, а для вычисления потери содержимого используется список слоев, включающий в себя нижние и верхние слои. Общая потеря вариации добавляется в конец.

В зависимости от используемых изображений с целевым содержимым и образцом стиля, может появиться желание настроить коэффициент content\_weight (определяет вклад потерь содержимого в общую величину потерь). Большее значение content\_weight обеспечит большее сходство сгенерированного изображения с целевым.

**2.7 Функция общей потери вариации**

outputs\_dict = dict([(layer.name, layer.output) for layer in model.layers]) #*Словарь, отображающий имена слоев в тензоры активаций*

content\_layer = 'block5\_conv2' #*Слой, используемый для вычисления потерь содержимого*

style\_layers = ['block1\_conv1',

'block2\_conv1',

'block3\_conv1',

'block4\_conv1',

'block5\_conv1'] #*Слой, используемый для вычисления потерь стиля*

total\_variation\_weight = 1e-4 #*Веса для вычисления среднего взвешенного по компонентам потерь*

style\_weight = 1.

content\_weight = 0.025

loss = K.variable(0.) #*Величина потерь определяется сложением всех компонентов с этой переменной*

layer\_features = outputs\_dict[content\_layer] #*Добавление потери содержимого*

target\_image\_features = layer\_features[0, :, :, :]

combination\_features = layer\_features[2, :, :, :]

loss += content\_weight \* content\_loss(target\_image\_features,

combination\_features)

for layer\_name in style\_layers: #*Добавление потери стиля для каждого целевого уровня*

layer\_features = outputs\_dict[layer\_name]

style\_reference\_features = layer\_features[1, :, :, :]

combination\_features = layer\_features[2, :, :, :]

sl = style\_loss(style\_reference\_features, combination\_features)

loss += (style\_weight / len(style\_layers)) \* sl

loss += total\_variation\_weight \* total\_variation\_loss

(combination\_image) # *Добавление общей потери вариации*

Наконец, настроим процесс градиентного восхождения. В оригинальной статье Гатиса оптимизация выполняется с использованием алгоритма L-BFGS, поэтому мы тоже используем его здесь. Реализация алгоритма L-BFGS уже включена в пакет SciPy, однако она имеет два незначительных ограничения:

* требует передачи значений функции потерь и градиентов в виде двух отдельных функций;
* может применяться только к плоским векторам, тогда как у нас используется

трехмерный массив с изображением.

Было бы неэффективно вычислять значения потерь и градиентов независимо, потому что это повлечет большой объем избыточных вычислений; процесс вычисления замедлится почти вдвое по сравнению со случаем, когда эти величины вычисляются вместе. Чтобы обойти эту проблему, определим класс Evaluator, вычисляющий значения потерь и градиентов одновременно, который будет возвращать значение потерь при первом обращении и кэшировать градиенты для повторного вызова.

**2.8 Подготовка процедуры градиентного спуска**

grads = K.gradients(loss, combination\_image)[0] #*#Получение градиентов сгенерированного изображения относительно потерь*

fetch\_loss\_and\_grads = K.function([combination\_image], [loss, grads]) # *Функция для получения значений текущих потерь и градиентов*

class Evaluator(object): #*Этот класс обертывает fetch\_loss\_and\_grads и позволяет получать потери и градиенты вызовами двух отдельных методов, как того требует реализация оптимизатора из SciPy*

def \_\_init\_\_(self):

self.loss\_value = None

self.grads\_values = None

def loss(self, x):

assert self.loss\_value is None

x = x.reshape((1, img\_height, img\_width, 3))

outs = fetch\_loss\_and\_grads([x])

loss\_value = outs[0]

grad\_values = outs[1].flatten().astype('float64')

self.loss\_value = loss\_value

self.grad\_values = grad\_values

return self.loss\_value

def grads(self, x):

assert self.loss\_value is not None

grad\_values = np.copy(self.grad\_values)

self.loss\_value = None

self.grad\_values = None

return grad\_values

evaluator = Evaluator()

Теперь можно запустить процесс градиентного восхождения с использованием реализации алгоритма L-BFGS в SciPy, сохраняя текущее сгенерированное изображение после каждой итерации алгоритма (в данном случае одной итерации соответствуют 20 шагов градиентного восхождения).

**2.9 Цикл передачи стиля**

from scipy.optimize import fmin\_l\_bfgs\_b

from scipy.misc import imsave

import time

result\_prefix = 'my\_result'

iterations = 20

x = preprocess\_image(target\_image\_path) # *Первичное состояние: целевое изображение*

x = x.flatten() #*Преобразуйте изображение, потому чтосscipy.optimize.fmin\_l\_bfgs\_b могут обрабатывать только плоские векторы*

for i in range(iterations):

print('Start of iteration', i)

start\_time = time.time()

x, min\_val, info = fmin\_l\_bfgs\_b(evaluator.loss,

x,

fprime=evaluator.grads,

maxfun=20)# *Выполняет оптимизацию L-BFGS по пикселам генерируемого изображения, чтобы минимизировать потерю стиля. Обратите внимание: вам нужно передать функцию, которая вычисляет потерю, и функцию, которая вычисляет градиенты, как два отдельных аргумента*

print('Current loss value:', min\_val)

img = x.copy().reshape((img\_height, img\_width, 3))

img = deprocess\_image(img)

fname = result\_prefix + '\_at\_iteration\_%d.png' % i

imsave(fname, img)

print('Image saved as', fname)

end\_time = time.time() . # *Сохраняет текущее сгенерированное изображение*

print('Iteration %d completed in %ds' % (i, end\_time - start\_time))

**3 Методы**

Данная работа — лишь одна из форм ретекстурирования изображений, или передачи текстуры. Лучшие результаты с его применением получаются, если изображения с образцами стилей сильно текстурированы и самоподобны, а целевые изображения с содержимым не требуют различения мелких деталей, чтобы их можно было опознать. Этот прием не наделен возможностями абстрагирования — с его помощью едва ли получится перенести стиль с одного портрета в другой. Данный алгоритм ближе

к классической обработке сигналов, чем к ИИ, поэтому не нужно ожидать от него чего-то сверхъестественного!

Кроме того, учтите, что этот алгоритм передачи стиля выполняется довольно медленно. Однако выполняемые преобразования достаточно просты, чтобы их можно было исследовать с использованием небольшой и быстрой сверточной сети при наличии достаточного объема обучающих данных. Быстрой передачи стиля можно достичь, если сначала потратить много времени на создание входных/выходных обучающих примеров для фиксированного изображения с образцом стиля, использовав метод, описанный здесь, а затем обучить простую сверточную сеть данному конкретному преобразованию стиля. После этого можно будет почти мгновенно стилизовать любое изображение: для этого потребуется просто пропустить его через эту маленькую сверточную сеть.

**5 Эксперименты**

Во второй главе описан метод переноса стиля с помощью библиотеки Keras данная библиотека была выбрана, так как давала самый лучший результат. Но были использованы и другие библиотеки: tensorflow и pytorch.

В данной работе так же были проведены эксперименты по созданию новых текстур:



Пример эффекта приложения Prizma

Перенос культурных стилей на логотипы:





Изменение гиперпараметров

В ходе каждого эксперимента отслеживались промежуточные результаты или результаты каждый итерации, для оценки качества работы модели. Для изменения результатов чаще всего менялось количество итераций и изменялись веса для потери передачи стиля (style\_weight) или изображения(content\_weight).

**5 Заключение**

Передача стиля заключается в создании нового изображения, которое сохраняет содержимое целевого изображения и оформлено в стиле изображения-образца.

Таким образом, передачу стиля в глубоком обучении можно сформулировать как процесс оптимизации, использующий функцию потерь, которая определяется предварительно обученной сверточной сетью.

Можно реализовать множество разнообразных вариантов для редактирования изображений

Библиотека keras и сеть VGG16 стала оптимальной для изучения нейронного переноса стиля в данной работе

Список литературы

[1] Shin H. C. et al. Deep convolutional neural networks for computer-aided detection: CNN architectures, dataset characteristics and transfer learning //IEEE transactions on medical imaging. – 2016. – Т. 35. – №. 5. – С. 1285-1298.

[2] Pan S. J., Yang Q. A survey on transfer learning //IEEE Transactions on knowledge and data engineering. – 2010. – Т. 22. – №. 10. – С. 1345-1359.

[3] Gatys L. A., Ecker A. S., Bethge M. Image style transfer using convolutional neural networks //Computer Vision and Pattern Recognition (CVPR), 2016 IEEE Conference on. – IEEE, 2016. – С. 2414-2423.

[4] Simonyan K., Zisserman A. Very deep convolutional networks for large-scale image recognition //arXiv preprint arXiv:1409.1556. – 2014.

[5] Li Y. et al. Demystifying neural style transfer //arXiv preprint arXiv:1701.01036. – 2017.

[6] Johnson J., Alahi A., Fei-Fei L. Perceptual losses for real-time style transfer and super-resolution //European Conference on Computer Vision. – Springer, Cham, 2016. – С. 694-711.

[7] Hecht-Nielsen R. Theory of the backpropagation neural network //Neural networks for perception. – 1992. – С. 65-93. [8] Ulyanov D. et al. Texture Networks: Feed-forward Synthesis of Textures and Stylized Images //ICML. – 2016. – С. 1349-1357.

[9] Ulyanov D., Vedaldi A., Lempitsky V. Improved texture networks: Maximizing quality and diversity in feed-forward stylization and texture synthesis //Proc. CVPR. – 2017.

[10] Ioffe S., Szegedy C. Batch normalization: Accelerating deep network training by reducing internal covariate shift //arXiv preprint arXiv:1502.03167. – 2015.

[11] Dumoulin V., Shlens J., Kudlur M. A learned representation for artistic style //CoRR, abs/1610.07629. – 2016. – Т. 2. – №. 4. – С. 5. 24

[12] Huang X., Belongie S. Arbitrary style transfer in real-time with adaptive instance normalization //CoRR, abs/1703.06868. – 2017.

[13] Zhang H., Dana K. Multi-style generative network for real-time transfer //arXiv preprint arXiv:1703.06953. – 2017.

[14] Gatys L. A. et al. Controlling perceptual factors in neural style transfer //IEEE Conference on Computer Vision and Pattern Recognition (CVPR). – 2017.

[15] Jing Y. et al. Stroke Controllable Fast Style Transfer with Adaptive Receptive Fields //arXiv preprint arXiv:1802.07101. – 2018.

[16] He K. et al. Deep residual learning for image recognition //Proceedings of the IEEE conference on computer vision and pattern recognition. – 2016. – С. 770-778. [17] Lin T. Y. et al. Microsoft coco: Common objects in context //European conference on computer vision. – Springer, Cham, 2014. – С. 740-755.

[18] Kingma D. P., Ba J. Adam: A method for stochastic optimization //arXiv preprint arXiv:1412.6980. – 2014.