ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»
**Московский институт электроники и математики им. А.Н. Тихонова**

# КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмизация и программирование»

Нейронная сеть, стилизующая изображения

образовательная программа «Информатика и вычислительная техника»

Работу выполнили
студенты группы БИВ201:
Москаленко Ярослав Александрович
Кануков Денис Михайлович
Научный руководитель:
Константинов Юрий Алексеевич

Москва 2021

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

**Московский институт электроники и математики им. А.Н. Тихонова**

**ЗАДАНИЕ**
**на курсовую работу бакалавра**

студентам группы БИВ201 Москаленко Ярославу Александровичу и Канукову Денису Михайловичу

1. Тема работы: Нейронная сеть, стилизующая изображения

2. Требования к выполнению:
Создать нейронную сеть, которая на основе анализа изображения будет переносить стиль на другое изображение.

**Требования к программной части:**

Работа должна быть реализована на языке программирования Python

**Содержание работы**
1. Введение
2. Описание алгоритма
3. Разработка ПО
4. Тестирование
5. Итоги
6. Список литературы

**Требование к программной документации**
1. Отчет;
2. Описание алгоритма решения поставленной задачи;
3. Описание кода;
4. Описание тестирования.

## Сроки выполнения этапов работы

Первый вариант КР предоставляется студентом в срок до «30» марта 2021г.

Итоговый вариант КР предоставляется студентом в срок до «30» мая 2021г.

| | | |
|---|---|---|
| Задание выдано | «20» декабря 2020 г. | Константинов Ю.А.<br>подпись руководителя |
| Задание принято к исполнению | «20» декабря 2020г. | Москаленко Я.А.<br>подпись студента |
| Задание принято к исполнению | «20» декабря 2020г. | Кануков Д.М.<br>подпись студента |

# Аннотация

Целью курсовой работы является создание нейросети, которая будет переносить стиль с одного изображения на второе. Степень стилизации можно регулировать путем изменения гиперпараметров. Данная нейросеть будет написана на языке программирования Python на основе предобученной сверточной нейронной сети VGG19.

Работа состоит из следующих глав:

- введение;
- описание алгоритма;
- разработка ПО;
- тестирование;
- итоги работы.

В введении описана актуальность выбранной темы курсовой работы, сформулирована задача, описана предметная область. Раздел «описание алгоритма» состоит из описания структуры нейронных сетей и алгоритма стилизации. Результат работы алгоритма представлен в главе «тестирование».

**Ключевые слова:** стилизация изображений, перенос стиля, Neural Style Transfer, Python.

# 1. Введение

В последние несколько лет мы испытали применение компьютерного зрения почти в каждом уголке нашей жизни — благодаря наличию огромных объемов данных и мощных графических процессоров, которые сделали обучение и развертывание сверточных нейронных сетей очень простым. Одна из самых интересных дискуссий сегодня в рамках машинного обучения заключается в том, как оно может повлиять и сформировать наше культурное и художественное производство в ближайшие десятилетия. Передача нейронного стиля - одно из самых креативных приложений сверточных нейронных сетей.

Сейчас многие компании внедряют в свои системы методы машинного обучения, а в частности для решения задач компьютерного зрения.

Несмотря на то, что на рынке существует несколько готовых решений, мы решили написать свой алгоритм, степень стилизации которого будет выше аналогов при этом не теряя контент изображения.

Целью курсовой работы являются изучение основных методов построения сверточной нейронной сети, методов работы с изображениями.

# 2. Описание алгоритма

## Сверточные нейронные сети

Сверточная нейронная сеть или CNN - это тип модели глубокого обучения для обработки данных, которая имеет сеточный шаблон, такой как изображения, который вдохновлен организацией зрительной коры животных и предназначен для автоматического и адаптивного изучения пространственных иерархий объектов, от низкоуровневых до высокоуровневых паттернов. CNN - это математическая конструкция, которая обычно состоит из трех типов слоев: свертка, объединение и полностью связанные слои. Первые два, слои свертки и объединения, выполняют извлечение объектов, в то время как третий, полностью связанный слой, отображает извлеченные объекты в конечный результат, такой как классификация. Слой свертки играет ключевую роль в CNN, который состоит из стека математических операций, таких как свертка, специализированный тип линейной операции. В цифровых изображениях значения пикселей хранятся в двумерной (2D) сетке, т. е. в массиве чисел. Пример на рис. 1

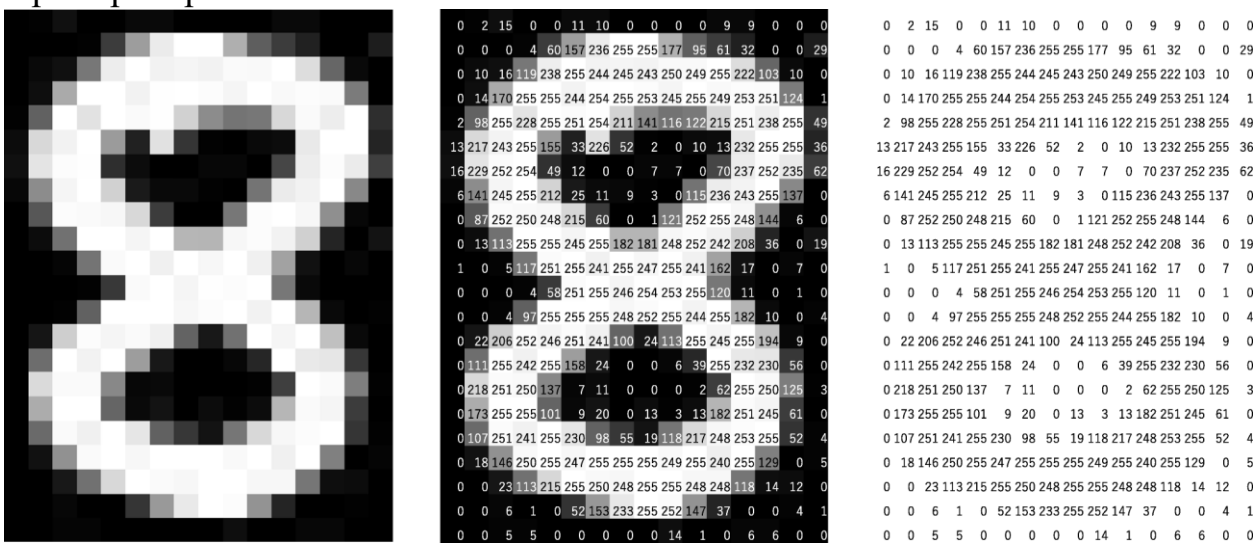


Рис.1

Компьютер видит изображение в виде массива чисел. Матрица справа содержит числа от 0 до 255, каждое из которых соответствует яркости пикселя на левом изображении. Оба они наложены на среднее изображение.

Каждый отдельный нейрон работает только в своем рецептивном поле и связан с другими нейронами так, что они охватывают все поле зрения. Подобно тому, как каждый нейрон реагирует на стимулы только в

ограниченной области - рецептивном полем в биологии, каждый нейрон в сверточной нейронной сети обрабатывает данные только в своем собственном рецептивном поле. Слои CNN расположены так, что первые слои обнаруживают более простые патерны, например: линии, кривые и т.д. Последние  слои обнаруживают более сложные патерны такие как лица, объекты и т.д.

## Архитектура сверточной нейронной сети

Архитектура CNN включает в себя несколько строительных блоков, таких как сверточные слои, слои объединения и полносвязанные слои. Типичная архитектура состоит из повторений стека из нескольких слоев свертки и слоя объединения, за которым следует один или несколько полносвязанных слоев. Шаг, на котором входные данные преобразуются в выходные через эти слои, называется прямым распространением (рис. 2). Хотя операции свертки и объединения, описанные в этом разделе, предназначены для 2D-CNN, аналогичные операции также могут быть выполнены для трехмерной 3D-CNN.

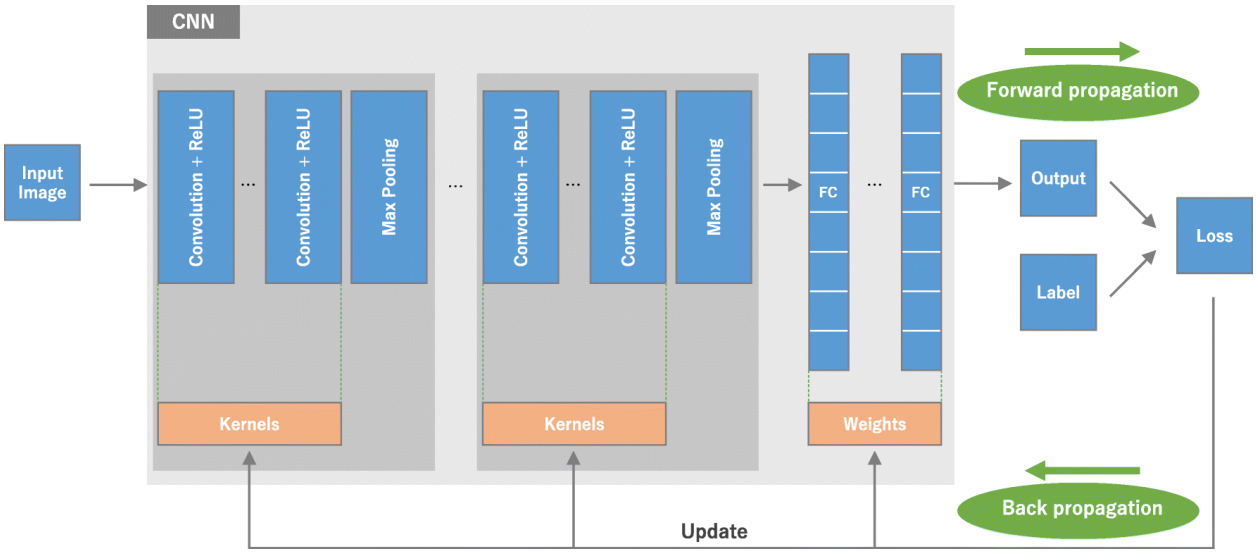
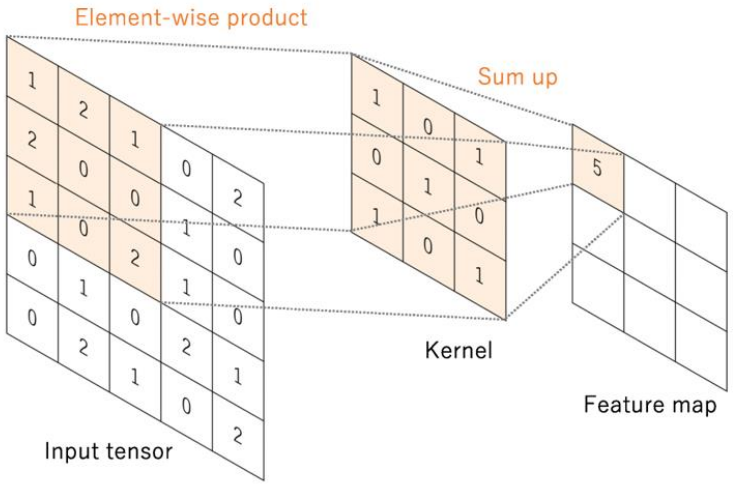


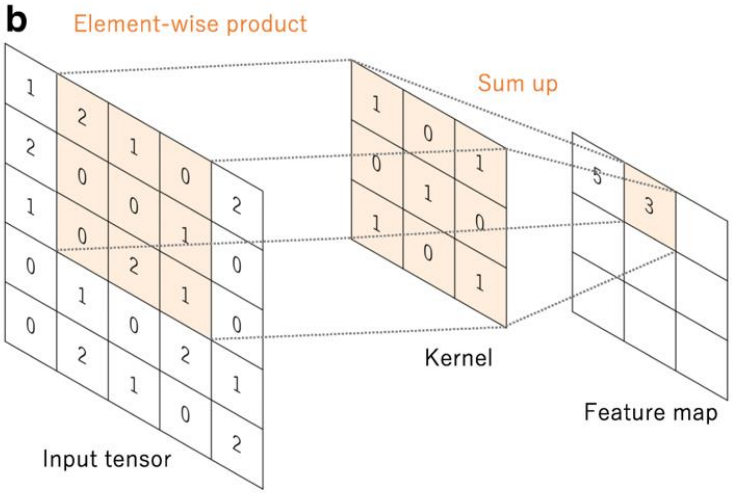

Рис.2

## Слой свертки

Свертка-это специализированный тип линейной операции, используемой для извлечения объектов, когда на входе применяется небольшой массив чисел, называемый ядром, который представляет собой массив чисел, называемый тензором. Поэлементное произведение между каждым элементом ядра и входным тензором вычисляется в каждом

местоположении тензора и суммируется для получения выходного значения в соответствующем положении выходного тензора, называемого картой объектов (рис. 3a-c). Эта процедура повторяется с применением нескольких ядер для формирования произвольного числа карт объектов, которые представляют различные характеристики входных тензоров; таким образом, различные ядра могут рассматриваться как различные экстракторы объектов. Двумя ключевыми гиперпараметрами, определяющими операцию свертки, являются размер и количество ядер. Первый, как правило, 3 × 3, но иногда 5 × 5 или 7 × 7. Последнее является произвольным и определяет глубину выходных карт объектов.



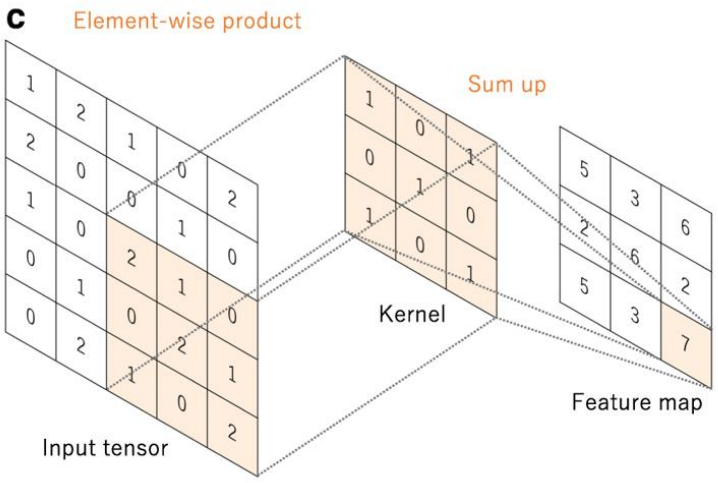



Рис.3

На рис.3(a–c) Пример операции свертки с размером ядра 3 × 3, без заполнения и шагом 1. Ядро применяется к входному тензору, и поэлементное произведение между каждым элементом ядра и входным тензором вычисляется в каждом местоположении и суммируется для получения выходного значения в соответствующей позиции выходного тензора, называемого картой объектов.

Операция свертки, описанная выше, не позволяет центру каждого ядра перекрывать самый внешний элемент входного тензора и уменьшает высоту и ширину карты выходных объектов по сравнению с входным тензором. Заполнение, обычно нулевое заполнение,-это метод решения этой проблемы, при котором строки и столбцы нулей добавляются с каждой стороны входного тензора, чтобы соответствовать центру ядра на самом внешнем элементе и сохранять одно и то же измерение в плоскости во время операции свертки (рис. 4). Современные архитектуры CNN обычно используют нулевое заполнение для сохранения размеров в плоскости, чтобы применить больше слоев. Без заполнения нуля каждая последующая карта объектов будет уменьшаться после операции свертки.

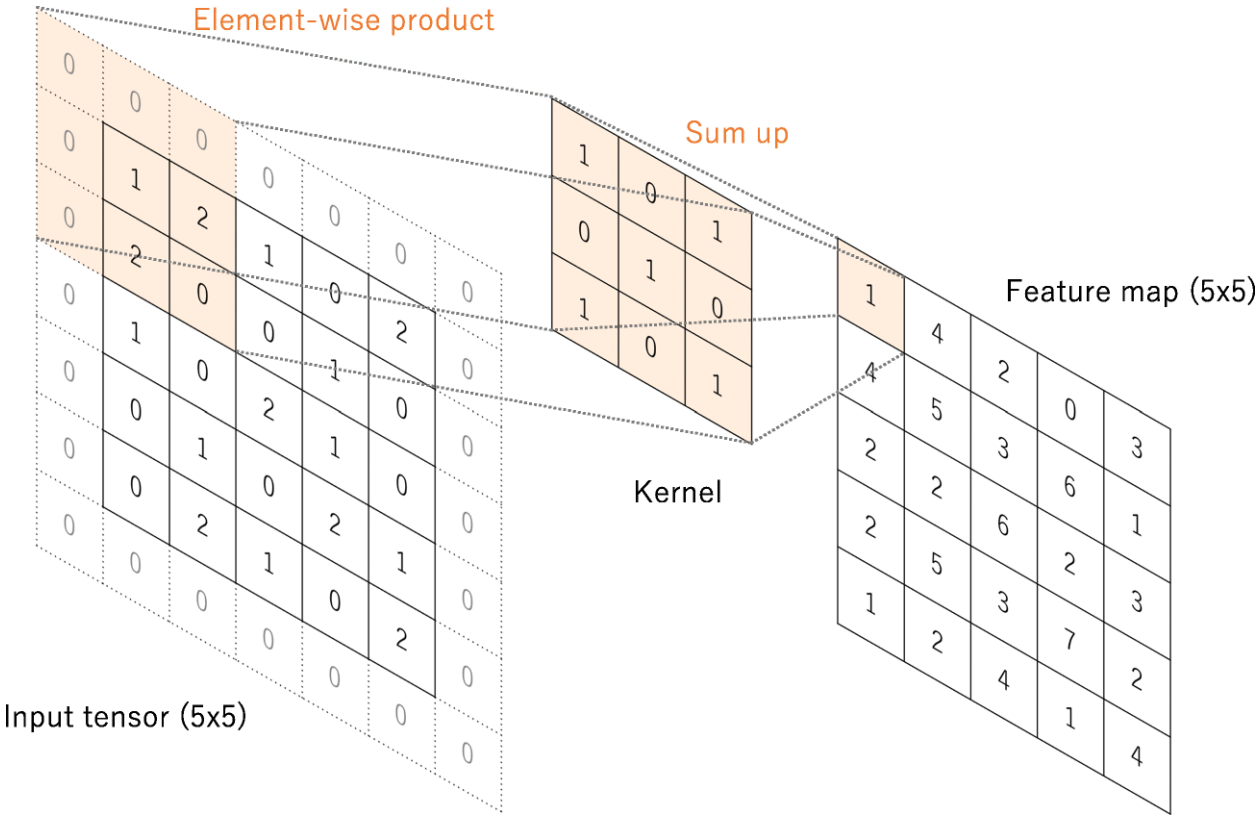
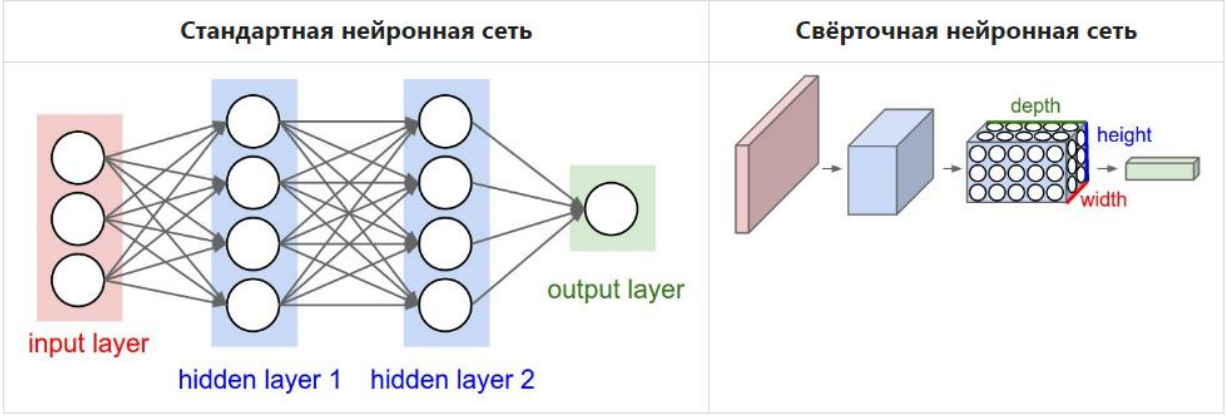



Рис.4

Таким образом, мы видим сильные стороны сверток: выделение более сложных патернов на каждом новом слою и уменьшение размерности данных, подаваемых в полно-связные слои, что повышает статистическую эффективность модели.

Процесс обучения модели CNN с учетом уровня свертки заключается в определении ядер, которые лучше всего работают для данной задачи на основе данного обучающего набора данных. Ядра-это единственные параметры, которые автоматически усваиваются в процессе обучения в слое свертки; с другой стороны, размер ядер, количество ядер, заполнение и шаг-это гиперпараметры, которые необходимо задать до начала процесса обучения



**Pooling-слой**

Pooling применяется после сверточных слоев в выборочных местах . и помогает уменьшить пространственный размер карты активации, что уменьшает необходимый объем вычислений и обновляемых весов.

Слой объединения обеспечивает типичную операцию понижающей дискретизации, которая уменьшает размерность карт объектов в плоскости, чтобы ввести инвариантность преобразования к небольшим сдвигам и искажениям и уменьшить количество последующих обучаемых параметров. Следует отметить, что ни в одном из слоев объединения нет обучаемого параметра, в то время как размер фильтра, шаг и заполнение являются гиперпараметрами в операциях объединения, аналогично операциям свертки.

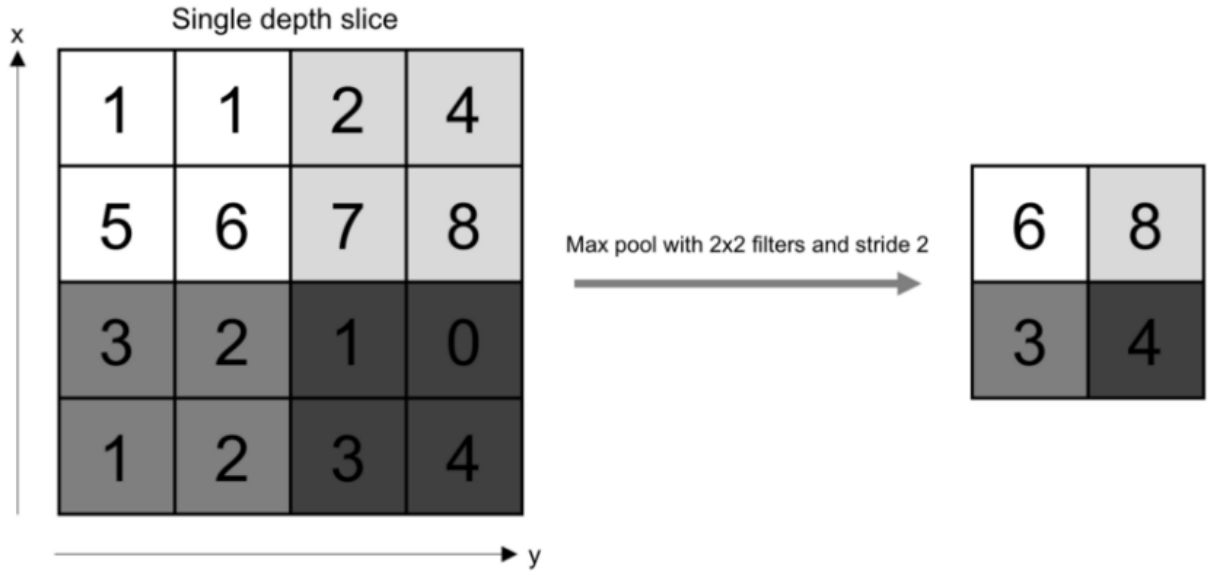



Рис.5

Наиболее популярной формой объединения является Max Pooling, который извлекает патчи из входных карт объектов, выводит максимальное значение в каждом патче и отбрасывает все остальные значения (рис. 5). На практике обычно используется максимальное объединение с фильтром размером 2 × 2 с шагом 2. Это уменьшило размерность карт объектов в плоскости в 2 раза. В отличие от высоты и ширины, размер глубины карт объектов остается неизменным.

**Полно-связные**

Нейроны в слое имеют полную связь с каждым нейроном в предыдущем и последующем слое, т.е каждый нейрон представляет из себя аналог линейной регрессии. Вот почему его можно вычислить, как обычно, путем умножения матрицы с последующим эффектом смещения.

Выходные карты объектов конечного слоя свертки или объединения обычно сглаживаются, т. Е. Преобразуются в одномерный (1D) массив чисел (или вектор) и соединяются с одним или несколькими полностью связанными слоями, также известными как плотные слои, в которых каждый вход соединен с каждым выходом с помощью обучаемого веса. После создания объектов, извлеченных слоями свертки и уменьшенных слоями объединения, они сопоставляются подмножеством полностью связанных слоев с конечными выходами сети, такими как вероятности для каждого класса в задачах классификации. Последний полностью подключенный слой обычно имеет такое же количество выходных узлов, как и количество классов. За каждым полностью связанным слоем следует нелинейная функция.

Нелинейная функция или функции активации бывают разных видов и подбираются для каждой конкретной задачи, но за самой универсальной принято считать ReLU

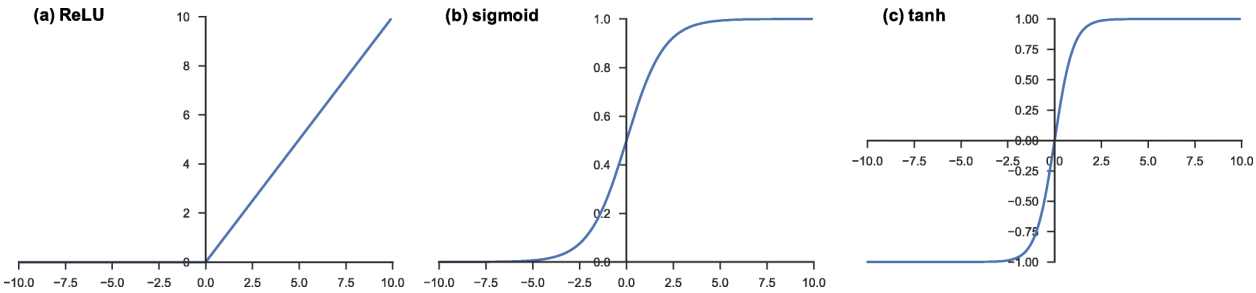



Рис.6

**Обучение сверточной нейросети**

**Функция потерь**

Функция потерь, также называемая функцией затрат, измеряет совместимость между выходными прогнозами сети посредством прямого распространения ошибки и заданными метками истинности. Обычно используемая функция потерь для многоклассовой классификации-это перекрестная энтропия, в то время как среднеквадратичная ошибка обычно применяется для регрессии к непрерывным значениям. Тип функции потерь является одним из гиперпараметров и должен определяться в соответствии с заданными задачами.

**Градиентный спуск**

Градиентный спуск обычно используется в качестве алгоритма оптимизации, который итеративно обновляет изучаемые параметры, т. Е. ядра и веса, сети, чтобы минимизировать потери. Градиент функции потерь дает нам направление, в котором функция имеет наибольшую скорость увеличения, и

каждый обучаемый параметр обновляется в отрицательном направлении градиента с произвольным размером шага, определяемым на основе гиперпараметра, называемого скоростью обучения (рис. 7). Градиент математически является частной производной потерь по каждому изучаемому параметру, и однократное обновление параметра формулируется следующим образом:

$$\mathrm{w} = \mathrm{w} - \alpha * \frac{\partial L}{\partial \mathrm{w}}$$

где w обозначает каждый обучаемый параметр, α обозначает скорость обучения, а L обозначает функцию потерь. Следует отметить, что на практике скорость обучения является одним из наиболее важных гиперпараметров, которые должны быть установлены до начала обучения.

Кроме того, было предложено и широко использовано множество улучшений алгоритма градиентного спуска, таких как SGD с импульсом, RMSProp и Adam, которым мы и воспользуемся в своей работе.

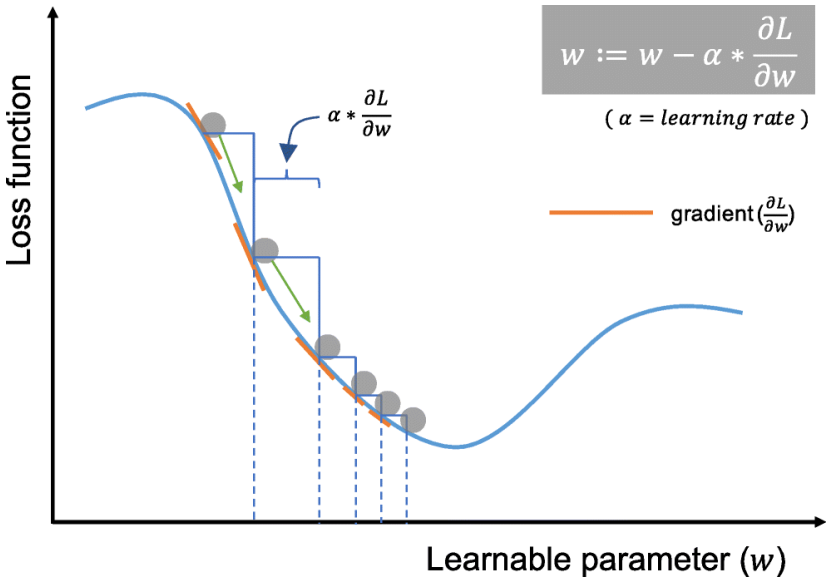



Рис. 7

# Алгоритм стилизации изображений

## Определение представлений контента и стиля

Чтобы получить представление как о содержании, так и о стиле нашего изображения, мы рассмотрим некоторые промежуточные слои в нашей модели. Промежуточные слои представляют собой карты объектов, которые становятся все более упорядоченными по мере углубления. В данном случае мы используем сетевую архитектуру VGG 19, предварительно подготовленную сеть классификации изображений. Эти промежуточные слои необходимы для определения представления контента и стиля на наших изображениях. Для входного изображения мы постараемся сопоставить соответствующие целевые представления стиля и контента на этих промежуточных слоях.

## Потеря контента

Определение потери контента довольно простое. Мы передадим сети как желаемое изображение контента, так и наше базовое входное изображение. Это вернет выходные данные промежуточного слоя (из слоев, определенных выше) из нашей модели. Затем возьмем евклидово расстояние между двумя промежуточными представлениями этих изображений.

Потеря контента - это функция, которая описывает расстояние контента от нашего входного изображения *x* и нашего изображения контента *p*. Формально описать потерю контента можно так:

$$L_{content}(p, x) = \sum_{i,j} \Big(F_{ij}(x) - P_{ij}(p)\Big)^2$$

## Потеря стиля

Вычисление потери стиля немного сложнее, но следует тому же принципу, на этот раз подавая нашей сети базовое входное изображение и изображение стиля. Однако вместо сравнения необработанных промежуточных выходов базового входного изображения и изображения стиля мы вместо этого сравниваем матрицы Грамма двух выходов.

Математически мы описываем потерю стиля базового входного изображения *x* и изображения стиля *a* как расстояние между представлением стиля (матрицами Грамм) этих изображений. Чтобы создать стиль для нашего базового входного изображения, мы выполняем градиентный спуск из изображения содержимого, чтобы преобразовать его в изображение, соответствующее представлению стиля исходного изображения. Мы делаем

это, минимизируя среднее квадратическое расстояние между картой корреляции объектов изображения стиля и входным изображением. Вклад каждого слоя в общую потерю стиля описывается следующим образом:

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{ij}^l - A_{ij}^l)^2$$

где $G_{ij}^l$ и $A_{ij}^l$-соответствующее представление стиля в слое $l$ входного изображения *x* и изображения стиля $a$. $N_l$ описывает количество карт объектов, каждая из которых имеет размер $M_l$ = высота*ширина. Таким образом, общая потеря стиля на каждом слое равна:

$$L_{style}(a, x) = \sum_{l \in L} \omega_l E_l$$

где мы взвешиваем вклад потерь каждого слоя по некоторому коэффициенту $\omega_l$. В нашем случае мы взвешиваем каждый слой одинаково:

$$\omega_l = \frac{1}{\|L\|}$$

**Градиентный спуск**

В этом случае мы используем оптимизатор Adam, чтобы минимизировать наши потери. Мы итеративно обновляем наше выходное изображение таким образом, чтобы оно минимизировало наши потери: мы не обновляем веса, связанные с нашей сетью, но вместо этого мы тренируем наше входное изображение, чтобы минимизировать потери.

Здесь мы используем tf.Gradient для вычисления градиента. Это позволяет нам воспользоваться преимуществами автоматического дифференцирования, доступного с помощью операций трассировки, для последующего вычисления градиента. Он записывает операции во время прямого прохода, а затем может вычислить градиент нашей функции потерь по отношению к нашему входному изображению для обратного прохода.

Таким образом, благодаря градиентному спуску мы итеративно обновляем наше выходное изображение, тем самым стилизуя его.

# 3. Разработка ПО

Импортируем необходимые библиотеки

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow import keras
from io import BytesIO
from PIL import Image
import cv2

%matplotlib inline
```

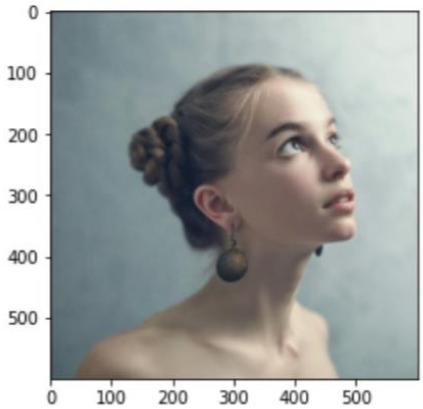
Загрузим изображения

```
upl = files.upload()
img = Image.open(BytesIO(upl['img.jpg']))
style = Image.open(BytesIO(upl['img_style.jpg']))
```

Выведем считанные изображения

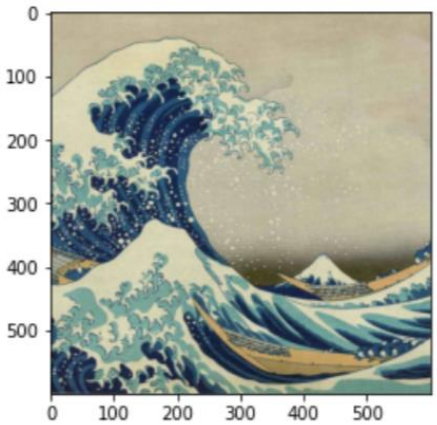
```
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x7fd620063e50>
```




```
plt.imshow(style)
```

```
<matplotlib.image.AxesImage at 0x7fd61ffcb690>
```




Преобразуем изображения в формат BGR, с которым работает CVV VGG1

```
x_img = tf.keras.applications.vgg19.preprocess_input(np.expand_dims(img, axis = 0))
x_style = tf.keras.applications.vgg19.preprocess_input(np.expand_dims(style, axis = 0))
```

Создадим функцию, которая будет переводить изображение из формата BGR обратно в формат RGB. Это понадобится нам далее.

```
def deprocess_img(processed_img):
    x = processed_img.copy()
    if len(x.shape) == 4:
        x = np.squeeze(x, 0)
    assert len(x.shape) == 3, ("Input to deprocess image must be an image of "
                               "dimension [1, height, width, channel] or [height, width, channel]")
    if len(x.shape) != 3:
        raise ValueError("Invalid input to deprocessing image")

    # perform the inverse of the preprocessing step
    x[:, :, 0] += 103.939
    x[:, :, 1] += 116.779
    x[:, :, 2] += 123.68
    x = x[:, :, ::-1]

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

Определим вспомогательные коллекции с названиями слоев, которые мы в дальнейшем будем выделять из сверточной нейронной сети.

```
content_layers = ['block5_conv2']


style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',
                'block5_conv1']

num_content_layers = len(content_layers)
num_style_layers = len(style_layers)
```

Загрузим модель VGG19, не используя полносвязную нейронную сеть на ее конце, а веса будем использовать предобученные и необучаемые.

```
vgg = keras.applications.vgg19.VGG19(include_top=False, weights='imagenet')
vgg.trainable = False
```

Выделим слои контента и стиля из нейронной сети VGG19

```
style_outputs = [vgg.get_layer(name).output for name in style_layers]
content_outputs = [vgg.get_layer(name).output for name in content_layers]
model_outputs = style_outputs + content_outputs
model = keras.models.Model(vgg.input, model_outputs)
```

Функция, которая возвращает карту признаков для стиля и контента

```
def get_feature_representations(model):
    style_outputs = model(x_style)
    content_outputs = model(x_img)

    style_features = [style_layer[0] for style_layer in style_outputs[:num_style_layers]]
    content_features = [content_layer[0] for content_layer in content_outputs[num_style_layers:]]
    return style_features, content_features
```

Функция, которая возвращает потери по контенту

```
def get_content_loss(base_content, target):
    return tf.reduce_mean(tf.square(base_content - target))
```

Функции, которые требуются для подсчета потери по стилю

```
def gram_matrix(input_tensor):
    channels = int(input_tensor.shape[-1])
    a = tf.reshape(input_tensor, [-1, channels])
    n = tf.shape(a)[0]
    gram = tf.matmul(a, a, transpose_a=True)
    return gram / tf.cast(n, tf.float32)

def get_style_loss(base_style, gram_target):
    height, width, channels = base_style.get_shape().as_list()
    gram_style = gram_matrix(base_style)
    return tf.reduce_mean(tf.square(gram_style - gram_target))
```

Функция, которая считает общие потери

```
def compute_loss(model, loss_weights, init_image, gram_style_features, content_features):
  style_weight, content_weight = loss_weights

  model_outputs = model(init_image)

  style_output_features = model_outputs[:num_style_layers]
  content_output_features = model_outputs[num_style_layers:]

  style_score = 0
  content_score = 0

  weight_per_style_layer = 1.0 / float(num_style_layers)
  for target_style, comb_style in zip(gram_style_features, style_output_features):
    style_score += weight_per_style_layer * get_style_loss(comb_style[0], target_style)

  weight_per_content_layer = 1.0 / float(num_content_layers)
  for target_content, comb_content in zip(content_features, content_output_features):
    content_score += weight_per_content_layer* get_content_loss(comb_content[0], target_content)

  style_score *= style_weight
  content_score *= content_weight

  loss = style_score + content_score
  return loss, style_score, content_score
```

Задаем гиперпараметры

```
num_iterations=7500
content_weight=1e3
style_weight=1e-2
```

Пропускаем через нейронную сеть VGG19 исходные изображения, чтобы получить их карты признаков. Сразу же вычисляем матрицу Грамма для изображения со стилем.

```
style_features, content_features = get_feature_representations(model)
gram_style_features = [gram_matrix(style_feature) for style_feature in style_features]
```

Создаем копию исходного изображения для дальнейшей стилизации

```
init_image = np.copy(x_img)
init_image = tf.Variable(init_image, dtype=tf.float32)
```

Создадим оптимизатор Adam. Вспомогательные переменные: счетчик цикла, лучшие потери, лучшее изображение, потери. Объеденим их в словарь для уобства использования.

```
opt = tf.compat.v1.train.AdamOptimizer(learning_rate=2, beta1=0.99, epsilon=1e-1)
iter_count = 1
best_loss, best_img = float('inf'), None
loss_weights = (style_weight, content_weight)

cfg = {
        'model': model,
        'loss_weights': loss_weights,
        'init_image': init_image,
        'gram_style_features': gram_style_features,
        'content_features': content_features
}
```

Переменные, для преобразования изображения из формата BGR в формат RGB. *imgs* будет содержать все изображения полученные в процессе работы модели.

```
norm_means = np.array([103.939, 116.779, 123.68])
min_vals = -norm_means
max_vals = 255 - norm_means
imgs = []
```

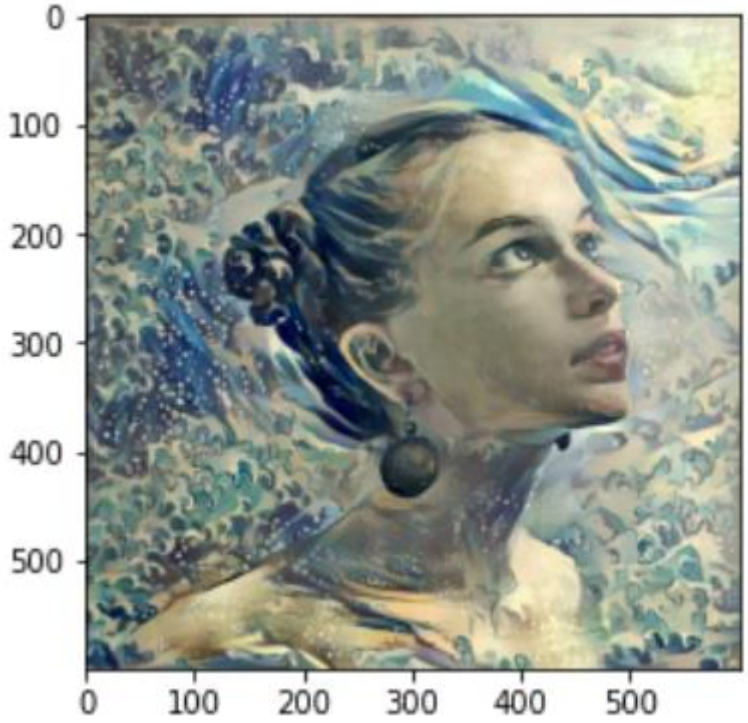
Запустим алгоритм градиентного спуска, стилизующий изображение

```
from tqdm import tqdm
for i in tqdm(range(num_iterations)):
    with tf.GradientTape() as tape:
        all_loss = compute_loss(**cfg)
    loss, style_score, content_score = all_loss
    grads = tape.gradient(loss, init_image)

    opt.apply_gradients([(grads, init_image)])
    clipped = tf.clip_by_value(init_image, min_vals, max_vals)
    init_image.assign(clipped)

    if loss < best_loss:
      best_loss = loss
      best_img = deprocess_img(init_image.numpy())


      plot_img = deprocess_img(init_image.numpy())
      imgs.append(plot_img)
```

Выведем полученное изображение и лучшие потери

```
plt.imshow(best_img)
print(best_loss)
```

```
tf.Tensor(177616.83, shape=(), dtype=float32)
```



# 4. Тестирование

## Исходные изображения:


Перебором гиперпараметров попробуем добиться лучшего результата (оптимальные гиперпараметры могут зависеть от выбора исходных изображений).

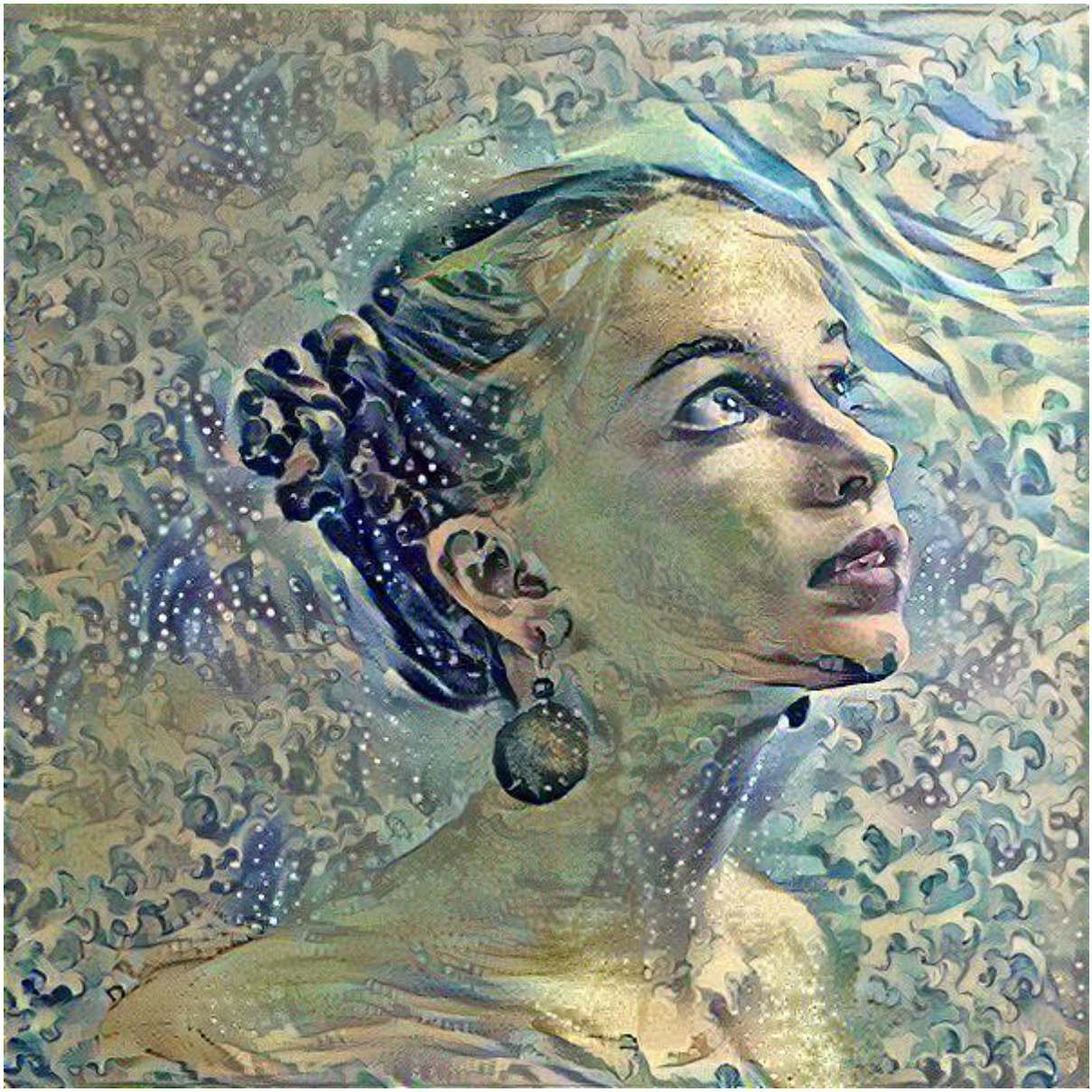
```
num_iterations=1000
content_weight=1e3
style_weight=1e-2
learning_rate=2
trainable = False
```

```
num_iterations=7500
content_weight=1e3
style_weight=1e-2
learning_rate=2
trainable = False
```

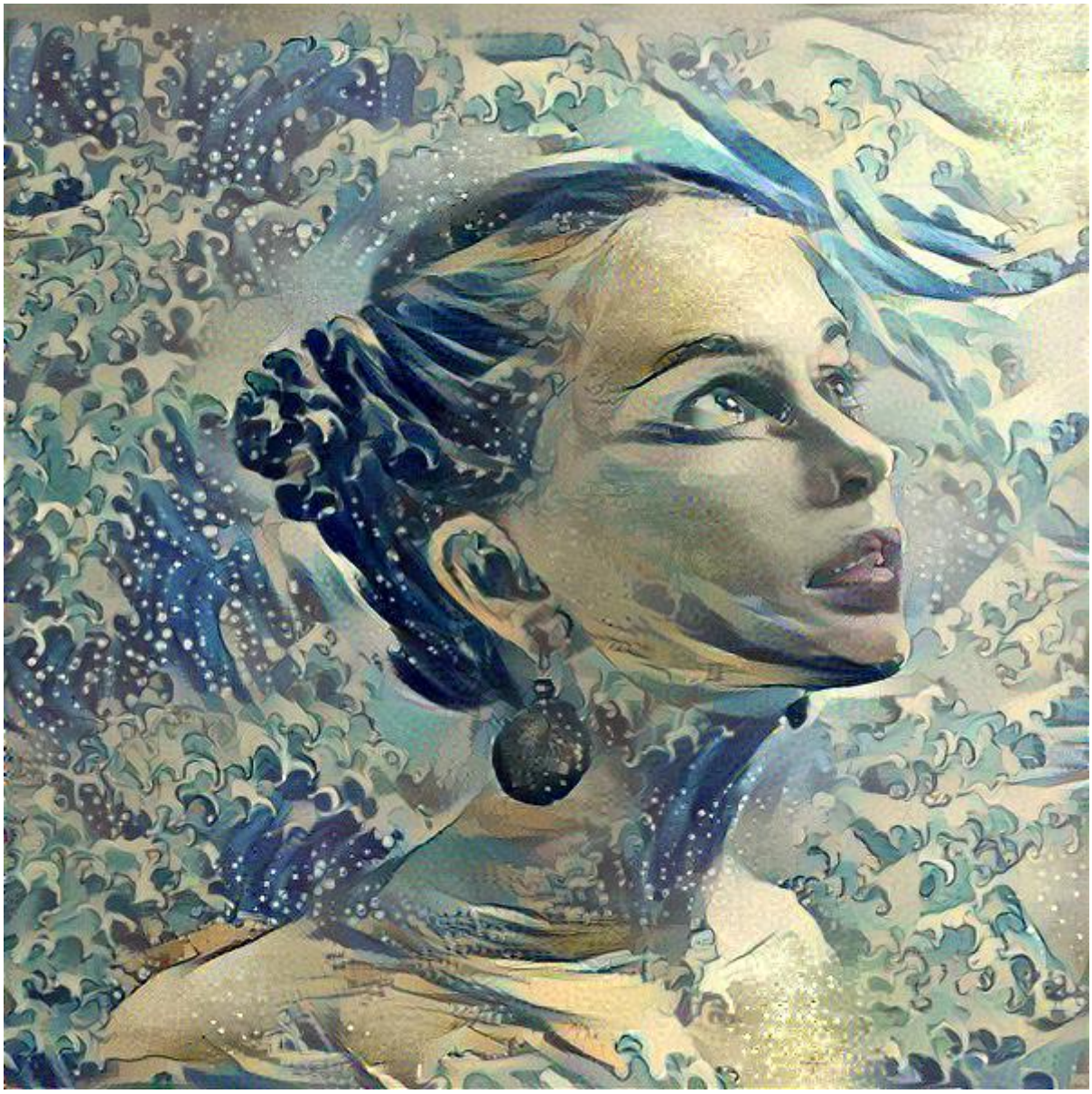
```
num_iterations=7500
content_weight=1e3
style_weight=1e-2
learning_rate=2
trainable = True
```

```
num_iterations=7500
content_weight=1e3
style_weight=1e-2
learning_rate=4
trainable = False
```

```
num_iterations=1000
content_weight=1e3
style_weight=1e-2
learning_rate=4
trainable = False
```



```
num_iterations=7500
content_weight=1e3
style_weight=1e-1
learning_rate=2
trainable = False
```



```
num_iterations=7500
content_weight=1e3
style_weight=1e-3
learning_rate=2
trainable = False
```

Наилучшими гиперпараметрами для этих исходных изображений оказались
`num_iterations=7500, content_weight=1e3, style_weight=1e-3, learning_rate=2, trainable = False`

# 5. Итоги работы

В процессе выполнения курсовой работой была изучена основная теория, связанная с построением сверточных нейронных сетей, обработки и стилизации изображений. На основе данного теоретического материала был разработан алгоритм переноса стиля изображений на базе предобученной сверточной нейронной сети VGG19.

Программа была разработана на языке программирования Python, с использованием библиотек matplotlib.pyplot, numpy, tensorflow, PIL.

# 6. Список источников

1. Принцип стилизации изображения.
URL:https://towardsdatascience.com/neural-style-transfer-on-real-time-video-with-full-implementable-code-ac2dbc0e9822

2. Обзор топологий глубоких сверточных нейронных сетей.
URL: https://habr.com/ru/company/mailru/blog/311706/

3. Принцип работы сверточной сети VGG19.
URL: https://neurohive.io/ru/vidy-nejrosetej/vgg16-model/

4. Алгоритм стилизации изображений.
URL: https://keras.io/examples/generative/neural_style_transfer/

5. Перенос нейронного стиля.
URL: https://www.tensorflow.org/tutorials/generative/style_transfer

6. Описание алгоритма NST.
URL: http://neerc.ifmo.ru/wiki/index.php?title=Neural_Style_Transfer

**Листинг программы:**

https://colab.research.google.com/drive/19yClDSLrvm435GDXWDuVyy1wz3tivLJJ#scrollTo=XLm4lysC6XYQ