Сжатие изображений

Тимур Мамедов, Борис Фаизов, Влад Шахуро

Graphics & Media Lab
Vision Group

В данный момент практически каждое изображение подвергается той или иной процедуре сжатия. При этом информация неизбежно теряется из изображений. Декодированные картинки должны быть как можно более похожи на исходные. В данном задании Вам предстоит разобрать два метода сжатия и декодирования изображений - PCA и JPEG. Вы сами реализуете данные методы и исследуете, как параметры методов влияют на различные метрики. Для удобства выполнения задание разделено на пункты. Примеры сжатия изображения на рисунке 1.

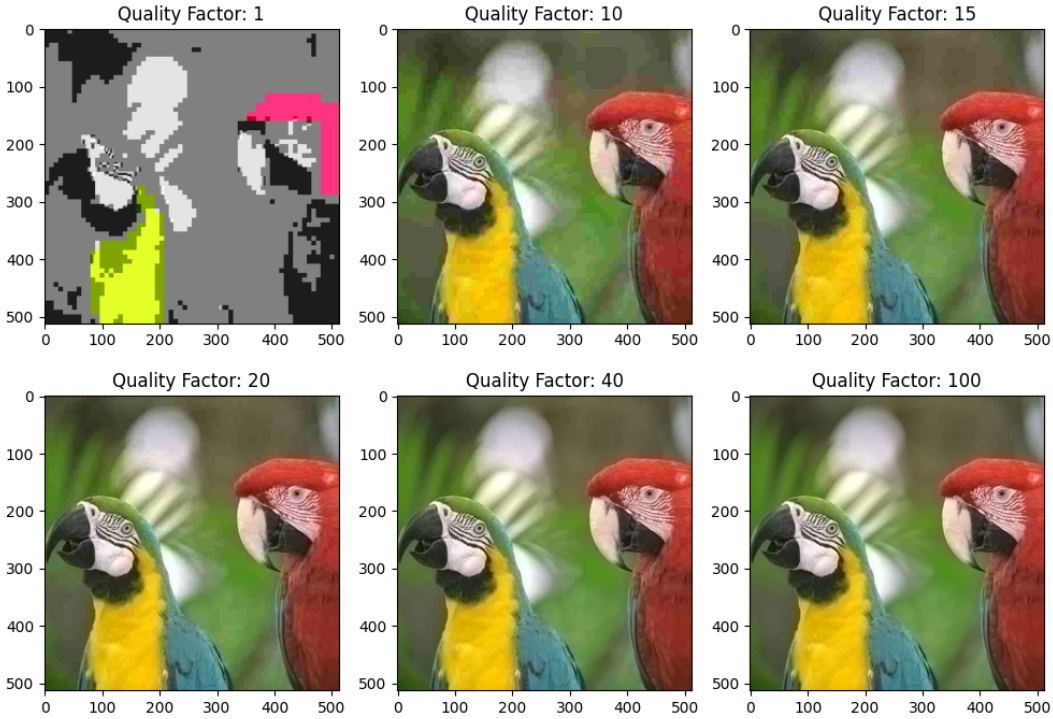

Рис. 1: Пример работы алгоритма JPEG.

1 Критерии оценки

Максимальная оценка за задание — 10 баллов. Баллы за каждый пункт задания указаны далее. Дополнительные вопросы (выделены цветными прямоугольниками в задании) никак не оцениваются.

2 Метод главных компонент PCA (3 балла)

Сначала кратко опишем суть метода. Его идея заключается в переходе из пространства размерности n в m-мерное пространство, где $m < n$. Например, рассмотрим черно-белую картинку размера 8×8 - его размерность, как вы уже могли догадаться, 64. Наша задача состоит в переходе в пространство

меньшей размерности и возвращении обратно. Конечно, во время сжатия будет происходить утрата информации, поэтому в итоге мы получим алгоритм *сжатия с потерями*.

Изображение из описания выше задается набором 64 значений пикселей, который мы опишем вектором $x = [x_1, x2, ..., x_{64}]$. Тогда картинку мы можем рассматривать в терминах пиксельного базиса. То есть $img(x) = x_1 \times (pixel_1) + x_2 \times (pixel_2) + ... + x_{64} \times (pixel_{64})$. Один из возможных способов понижения размерности - обнуление большей части базисных векторов. Очевидно, что из этого особо ничего хорошего не выйдет, мы потеряем слишком много информации и восстановить изображение корректно не получится.

Однако попиксельное представление - это не единственный вариант базиса. Возможно и использование других базисных функций, когда каждый пиксель вносит в каждую из них некий заранее определенный вклад: $img(x) = average + x_1 \times (basis_1) + x_2 \times (basis_2) + ...$ Метод PCA можно рассматривать как процесс выбора оптимальных базисных функций, таких, чтобы комбинации лишь нескольких из них было достаточно для удовлетворительного воссоздания основной части элементов картинки. Главные компоненты, служащие низкоразмерным представлением данных, будут в этом случае просто коэффициентами, умножаемыми на каждый из элементов ряда.

Подробнее про PCA можно прочитать здесь или здесь.

2.1 Сжатие изображения с помощью PCA (2 балла)

Главные компоненты строятся таким образом, чтобы они учитывали максимально возможную дисперсию в объекте. Как мы можем построить первую главную компоненту? Для этого можно взять линию, в направлении которой проекция точек имеет наибольшую дисперсию. Вторая главная компонента рассчитывается таким же образом. Берется второе направление с наибольшей дисперсией при условии, что оно не коррелировано (т.е. перпендикулярно) первой главной компоненте. Это продолжается до тех пор, пока не будет вычислено p главных компонент.

За всем этим стоят собственные векторы и собственные значения матрицы ковариации, потому что собственные векторы матрицы ковариации на самом деле являются направлениями осей, по которым наблюдается наибольшая дисперсия (большая часть информации) и которые мы называем главными компонентами. А собственные значения - это просто коэффициенты, связанные с собственными векторами, которые дают величину дисперсии, переносимую в каждом основном компоненте.

Ранжируя собственные векторы в порядке их собственных значений, от наибольшего к наименьшему, мы можем получить главные компоненты в порядке их значимости.

Напишите функцию `pca_compression(matrix, p)`, которая для данной двумерной матрицы $matrix$ (одна цветовая компонента картинки) и данного количества компонент p вернет:

- собственные векторы
- проекцию матрицы на новое пространство
- Средние значения, использованные при центрировании каждой строчки матрицы

Реализацию можно разделить на несколько этапов:

1. Центрирование каждой строчки матрицы (то есть нужно считать среднее значение в каждой строчке квадратной матрицы, $np.mean(M, axis = 1)[:, None]$).

$$M_{i,j} = M_{i,j} - \overline{M_i}$$

2. Поиск матрицы ковариации (COV - выборочная ковариация).

$$C = (COV(M_i, M_j))_{i,j}$$

3. Поиск собственных значений (eig_val) и собственных векторов (eig_vec) матрицы ковариации, используйте $linalg.eigh$ из $numpy$
4. Посчитаем количество найденных собственных векторов (это количество столбцов в матрице eig_vec из прошлого шага).
5. Сортируем собственные значения eig_val в порядке убывания при помощи $np.argsort$.
6. Сортируем собственные векторы eig_vec согласно отсортированным собственным значениям. Это все для того, чтобы мы производили проекцию в направлении максимальной дисперсии.
7. Оставляем только p первых собственных векторов (то есть делаем срез матрицы eig_vec, в котором оставляем первые p столбцов).
8. Проекция данных на новое пространство. Для этого скалярно умножим каждый оставшийся собственный вектор eig_vec на каждый стобец матрицы M. Это можно сделать при помощи перемножения матриц сразу для всех собственных векторов как $eig\_vec^T \times M$.

Проверьте реализацию с помощью юнит-тестов:

```
$ ./run.py unittest pca_compression
```

2.2 Восстановление изображения с помощью PCA (1 балл)

Напишите функцию `pca_decompression(compressed)`, которая принимает на вход список *compressed* из 3х кортежей для каждого канала. Каждый кортеж состоит из собственных векторов, проекций для каждой цветовой компоненты и из средних значений. То есть то, что вернула `pca_compression` для каждого канала. Функция должна вернуть "разжатое" изображение.

Сама операция очень простая. Она состоит в том, что мы матрично умножаем собственные векторы на проекции и прибавляем среднее значение по строкам исходной матрицы.

Проверьте реализацию с помощью юнит-тестов:

```
$ ./run.py unittest pca_decompression
```

2.3 Визуализация

Визуализируйте, как будет выглядеть картинка при разном количестве компонент. Для этого допишите функцию `pca_visualize()`. Сохраните результат в картинке "pca_visualization.png".

3 JPEG (6 баллов за реализацию всего метода)

3.1 Процедура кодирования

Первая часть данного подзадания заключается в реализации процедуры сжатия изображения. В конце мы должны получить изображение, занимающее меньший объем памяти.

3.1.1 Переход из одного пространства в другое и обратно (0.5 баллов)

Первым этапом в JPEG-сжатии является переход из пространства RGB в пространство YCbCr

Из RGB в YCbCR:

$$\begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} + \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.1687 & -0.3313 & 0.5 \\ 0.5 & -0.4187 & -0.0813 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Из YCbCR в RGB:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.402 \\ 1 & -0.34414 & -0.71414 \\ 1 & 1.77 & 0 \end{bmatrix} \cdot \begin{bmatrix} Y \\ C_b - 128 \\ C_r - 128 \end{bmatrix}$$

Напишите функции `rgb2ycbcr(image)` и `ycbcr2rgb(image)`. Первая для переход из пространства RGB в пространство YCbCr, а вторая для перехода из пространства YCbCr в пространство RGB.

Проверьте реализацию с помощью юнит-тестов:
`$ ./run.py unittest rgb2ycbcr2rgb`

3.1.2 Размытие цвета и яркости

Факт, являющийся основополагающим в JPEG-сжатии, - это то, что человеческое зрение более чувствительно к изменению яркости, нежели к изменению цвета. Давайте проверим эту гипотезу (скорее, аксиому)!

Важно! Везде используется гауссов фильтр с радиусом 10 из *scipy.ndimage.filters*!

В этом задании Вам нужно написать две функции:

1. Компоненту яркости Y оставим такой, какая она есть. Цветовые компоненты Cb и Cr размоем с помощью фильтра Гаусса. Объединим все компоненты в одну картинку и переведем ее обратно в RGB. Сохраните результат в картинке “gauss_1.png”. Функцию для построения визуализации назовите как `get_gauss_1()`.
2. Компоненту яркости Y размоем с помощью фильтра Гаусса. Цветовые компоненты Cb и Cr оставим такими, какие они есть. Объединим все компоненты в одну картинку и переведем ее обратно в RGB. Сохраните результат в картинке “gauss_2.png”. Функцию для построения визуализации назовите как `get_gauss_2()`.

Если вы в обоих случаях использовали одни и те же параметры в фильтре Гаусса, и в первом случае получили картинку, слабо отличающуюся от исходной, а во втором - напротив, то поздравляем - вы экспериментально доказали анатомический факт!

3.1.3 Уменьшение цветовых компонент (0.5 балла)

Следующий шаг в алгоритме JPEG - уменьшение разрешения цветовых компонент.

Вопрос: во сколько раз нужно уменьшить цветовые компоненты?

Ответ: подобный параметр подбирается везде по-разному. Мы будем уменьшать цветовые компоненты в 2 раза.

Вопрос: как именно уменьшать?

Ответ: в этом разделе рассмотрим следующий прием:

1. Размываем цветовые компоненты с помощью фильтра Гаусса с радиусом 10 из *scipy.ndimage.filters*.
2. Убираем каждую вторую строчку и столбец

Напишите функцию `downsampling(component)`. На вход она принимает цветовую компоненту *component*. Она возвращает уменьшенную в 2 раза по по ширине и по высоте компоненту.

Проверьте реализацию с помощью юнит-тестов:

```
$ ./run.py unittest downsampling
```

3.1.4 Деление компонент на блоки

Далее изображение делится на блоки равного размера. Чаще всего картинку делят на квадраты размера 8×8. Это связано с тем, что блок должен содержать более-менее однородную информацию, например, только часть неба или какой-нибудь текстуры.

Для ускорения программы мы будем делить компоненты на блоки и производить над ними операции, которые будут описаны ниже, одновременно. Поэтому сам процесс деления будет необходимо реализовать чуть позже в общем цикле.

3.1.5 Дискретное косинусное преобразование (0.5 баллов)

Дискретное косинусное преобразование производится по следующей формуле:

$$G_{u,v} = \frac{1}{4}\alpha(u)\alpha(v)\sum_{x=0}^{7}\sum_{y=0}^{7} g_{x,y}\cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right],$$

где g - матрица до ДКП, G - после ДКП. $\alpha(u) = \frac{1}{\sqrt{2}}$, если $u = 0$, иначе $\alpha(u) = 1$.

Напишите функцию `dct(block)`. На вход она принимает блок *block* размера 8×8. Она выполняет его дискретное косинусное преобразование и возвращает блок размера 8×8 после ДКП.

Проверьте реализацию с помощью юнит-тестов:

```
$ ./run.py unittest dct
```

3.1.6 Квантование (0.5 баллов)

Человеческий глаз очень хорошо улавливает низкие частоты, а вот высокие - нет. Поэтому появилась следующая мысль: почему бы не удалить высокие частоты? Действительно, сокращая их, мы не очень сильно портим картинку, но при этом ее объем может быть сокращен в несколько раз. Именно для этого и применяют квантование.

Идея квантования: каждый элемент каждой компоненты мы делим на некое число, а потом округляем результат. После столь нехитрой операции мы получим очень много нулей, которые будут характеризовать высокие частоты. Ноль не несет в себе особой информации, поэтому мы его можем смело удалить (об это подробнее будет описано ниже). Таким образом, объем картинки сократился в разы. Именно на этом этапе происходит потеря иформации. Наверное, вы уже догадались почему алгоритм сжатия JPEG - это *сжатие с потерями*.

Как выбрать число, на которое будем делить? Существуют различные так называемые матрицы квантования. В выданном коде задания приведены 2 варианта "основных" матриц $y\_quantization\_matrix$, $color\_quantization\_matrix$. Первая - матрица квантования яркости, а вторая - матрица квантования цвета. Для тех, кто хочет провести больше экспериментов с алгоритмом JPEG, чуть ниже будет предложено самим сформировать матрицы квантования.

Напишите функцию `quantization(block, quantization_matrix)`. На вход она принимает блок *block* размера 8×8 после применения ДКП и матрицу квантования $quantization\_matrix$. Она выполняет квантование и возвращает блок размера 8x8 после квантования. Округление осуществляем с помощью *np.round*.

Проверьте реализацию с помощью юнит-тестов:

```
$ ./run.py unittest quantization
```

3.1.7 Самостоятельная генерация матрицы квантования (0.5 балла)

Введем так называемый Quality Factor $Q \in [1, 100]$, с помощью него мы сможем "управлять" процессом сжатия. Q задается пользователем, а на его основании вычисляется Scale Factor S следующим образом:

$$S = \begin{cases} \frac{5000}{Q} & 1 \leq Q < 50 \\ 200 - 2Q & 50 \leq Q \leq 99 \\ 1 & Q = 100 \end{cases}$$

После этого берутся "основные" матрицы квантования, которые были приведены чуть выше, и пересчитываются их коэффициенты следующим образом:

$$Q_{i,j} = \lfloor \frac{50 + S \times D_{i,j}}{100} \rfloor,$$

где $D_{i,j}$ - значение $[i, j]$ элемента в "основной" матрице. Обратите внимание на округление вниз в формуле. Как вы уже могли догадаться, при $Q = 50$ мы будем иметь матрицы, равные "основным".

Подробнее можно прочитать здесь (см. п.2.1).

Напишите функцию `own_quantization_matrix(default_quantization_matrix, q)`. На вход она принимает блок "стандартную" матрицу квантования $default\_quantization\_matrix$ и Quality Factor q. Она выполняет генерацию матрицы квантования по Quality Factor и возвращает новую матрицу квантования.

Hint: если после проделанных операций какие-то элементы обнулились, то замените их единицами

Проверьте реализацию с помощью юнит-тестов:

```
$ ./run.py unittest own_quantization
```

3.1.8 Зигзаг-сканирование и сжатие (1 балл)

Сначала необходимо обойти полученный массив (блок) специальным образом (рисунок 2):

Далее начинается самое интресное - сжатие. После зигзаг-сканирования мы получили одномерный массив. Теперь нам нужно его каким-то образом сжать. Как было сказано выше, после квантования у нас должно получиться много нулей, от них нужно как-то избавиться. Для этого вместо последовательности из множества нулей мы вписываем 1 нуль и после него число, обозначающее их количество в последовательности. Пример того, как это работает на рисунке 3.

Напишите функции `zigzag(block)` и `compression(zigzag_list)`. Первая функция выполняет Зигзаг-сканирование. Она принимает на вход блок *block* размера 8×8. На выход она выдает список из элементов входного блока, получаемый после его обхода зигзаг-сканированием. Вторая функция

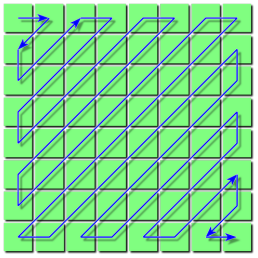

Рис. 2: Пример zigzag-обхода блока

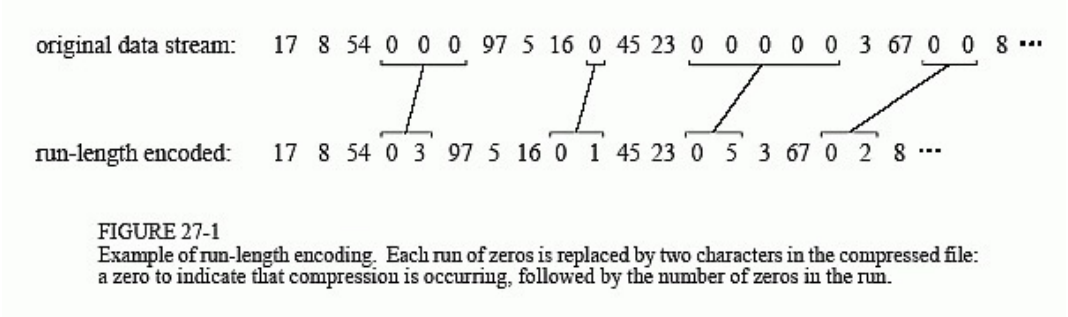

Рис. 3: Пример работы сжатия

выполняет сжатие последовательности после зигзаг-сканирования. Она получает на вход список после зигзаг-сканирования *zigzag_list*. На выходе она выдает сжатый список в формате, который был приведен в качестве примера на рисунке 3.

Проверьте реализацию с помощью юнит-тестов:

```
$ ./run.py unittest zigzag
$ ./run.py unittest compression
```

3.1.9 Итоговое сжатие изображения алгоритмом JPEG (баллы могут быть выставлены только после проверки декодирования)

Теперь необходимо все написанные выше функции соединить воедино.

Алгоритм:

1. Переводим картинку в YCbCr
2. Уменьшаем цветовые компоненты
3. Делим все компоненты на блоки 8x8 и все элементы блоков переводим из $[0, 255]$ в $[-128, 127]$
4. Ко всем блокам последовательно применяем дискретное косинусное преобразование, квантование (если вы реализовали свои матрицы квантования, то можете использовать их), зигзаг-сканирование и сжатие
5. Сохраняем в списки полученные сжатые данные

Напишите функцию `jpeg_compression(img, quantization_matrixes)`. Она принимает на вход цветную картинку *img* и список из 2-ух матриц квантования *quantization_matrixes*. На выходе из функции требуется получить список списков со сжатыми векторами: [[compressed_y1,...], [compressed_Cb1,...], [compressed_Cr1,...]].

3.2 Процедура декодирования изображения

Отлично, половина пути пройдена! В предыдущей части мы получили сжатые блоки компонент изображения. Теперь задача заключается в декодировании этих блоков и возвращении к исходному изображению. В итоге мы должны получить практически такое же изображение, которое было до процедуры сжатия.

3.2.1 Обратное сжатие и зигзаг-сканирование (1 балл)

Здесь и далее для разжатия изображения необходимо выполнить в обратном порядке обратные операции к тем, которые были применены в разделе "Процедура кодирования". Начнем с обратного сжатия и зигзаг-сканирования.

Суть первого заключается в том, что из короткого вектора, где последовательности из нулей были заменены на пары вида < 0, длина нулевой последовательности$>$, необходимо обратно получить вектор «полной длины», т.е. указанные пары меняются на последовательности нулей указанной длины.

Вторая операция чуть сложнее - необходимо от полученного на предыдущем шаге вектора перейти к двумерной матрице, т.е. расставить его элементы в матрице в порядке их следования в зигзаг-сканировании.

Напишите функции `inverse_compression(compressed_list)` и `inverse_zigzag(input)`. Первая функция принимает на вход сжатый список *compressed_list*. Возвращает функция "разжатый список". Вторая функция принимает на вход список элементов *input*. Возвращает блок размера 8×8 из элементов входного списка, расставленных в матрице в порядке их следования в зигзаг-сканировании.

Проверьте реализацию с помощью юнит-тестов:

```
$ ./run.py unittest inverse_compression
$ ./run.py unittest inverse_zigzag
```

3.2.2 Обратное квантование и дискретное косинусное преобразование (2 балла)

Суть обратного квантования заключается в домножении поданной на вход матрицы на матрицу квантования (если в кодировании вы использовали свои матрицы квантования, то при конечном декодировании не забудьте использовать именно их).

Обратное дискретное косинусное преобразование производится по следующей формуле:

$$f_{x,y} = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} \alpha(u)\alpha(v) F_{u,v} \cos\left[\frac{(2x+1)u\pi}{16}\right] \cos\left[\frac{(2y+1)v\pi}{16}\right],$$

где F - матрица до обратного ДКП, f - после обратного ДКП. $\alpha(u) = \frac{1}{\sqrt{2}}$, если $u = 0$, иначе $\alpha(u) = 1$. Не забываем округлять результат обратного ДКП, т.к. пиксели имеют целые значения из диапазона $[0, 255]$.

Напишите функции `inverse_quantization(block, quantization_matrix)` и `inverse_dct(block)`. Первая функция выполняет обратное квантование. Она принимает на вход блок размера 8×8 *block* после применения обратного зигзаг-сканирования и матрицу квантования *quantization_matrix*. Возвращает функция блок размера 8×8 "до" квантования. Округление не производится. Вторая фунцкия выполняет обратное дискретное косинусное преобразование. Она принимает на вход блок *block* размера 8×8. Возвращает блок размера 8×8 после обратного ДКП. Округление осуществляется помощью *np.round*.

Проверьте реализацию с помощью юнит-тестов:
```
$ ./run.py unittest inverse_quantization
$ ./run.py unittest inverse_dct
```

3.2.3 Увеличение цветовых компонент (0.5 баллов)

Как вы помните, в самом начале нашего пути мы уменьшили цветовые компоненты в 2 раза. Пришло время вернуть все как было. Для этого будем просто дублировать все пиксели: рассмотрим i-ю строчку, в которой был удален j-ый элемент, восстановим его значение с помощью значения элемента в позиции $(i, j-1)$; таким образом восстанавливаем i-ю строчку, далее, например, нами была удалена $(i+1)$-я строчка, тогда в нее дублируем $(i-1)$-ю строчку, и так далее.

Напишите функцию `upsampling(component)`. Она увеличивает цветовые компоненты в 2 раза. На вход она принимает цветовую компоненту *component* размера $[A, B]$. На выходе она возвращает цветовую компоненту размера $[2 * A, 2 * B]$.

3.2.4 Итоговое декодирование изображения алгоритмом JPEG

Все, что осталось - это все написанные выше функции соединить воедино.

Алгоритм:

1. Для всех векторов из result последовательно применяем следующие функции: обратное сжатие и зигзаг-сканирование, обратное квантование (если вы использовали свои матрицы квантования, то на этом этапе нужно применять именно их) и дискретное косинусное преобразование
2. После первого шага для каждой компоненты будем иметь набор блоков. Соединяем все эти блоки в 3 компоненты (не забываем значения каждого блока обратно переводить из $[-128, 127]$ в $[0, 255]$), т.е. в итоге должны получить полноразмернную компоненту яркости и две компоненты цвета, уменьшенные в 2 раза
3. Обе цветовые компоненты увеличиваем в 2 раза
4. Все компоненты объединяем в одно YCbCr изображение
5. Полученное изображение переводим обратно в RGB
6. Выводим результат

Напишите функцию `jpeg_decompression(result, result_shape, quantization_matrixes)`. Она принимает на вход список сжатых данных *result*, размер изображения-ответа *result_shape* и список из 2-ух матриц квантования. Возвращает функция разжатое изображение.

Визуализируйте, как будет выглядеть картинка при разном параметре Quality Factor. Для этого допишите функцию `jpeg_visualize()`. Сохраните результат в картинке “jpeg_visualization.png”.

3.3 Дополнительные эксперименты (не оценивается)

?

Для тех, кому больше всего понравилось это задание, предлагается провести еще несколько дополнительных экспериментов:

1. Посчитайте объем сжатой картинки (естественно, нужно вычислять объем тех данных, которые были получены в конце раздела “Процедура кодирования”). Насколько изменился вес картинки?
2. Поэкспериментируйте со значениями Quality Factor. Как будет выглядеть изображение при $Q = 1$, $Q = 100$?
3. Как значение Quality Factor влияет на объем сжатой картинки?

4 Общий Pipeline

В коде для Вашего удобства представлен общий *Pipeline* в функции `compression_pipeline(img, c_type, param)`. Это сделано сугубо для удобства запуска методов сжатия. В коде не нужно ничего редактировать. Если ваши функции соответствуют соглашениям, описанным в задании, то априори все должно работать корректно. Вход: исходное изображение *img*; название метода *c_type* - 'pca', 'jpeg'; *param* - кол-во компонент в случае PCA, и Quality Factor для JPEG. На выходе из функции возвращается изображение и количество бит на пиксель.

5 Метрики

Как и в любом другом вопросе, в сжатии также принято сравнивать те или иные методы с помощью определенных метрик. В этом разделе мы посчитаем PSNR для PCA и JPEG при разных значениях Quality Factor и количестве компонент соответственно, а также Rate-Distortion метрику, где в качестве Rate будет выступать количество бит на пиксель, а Distortion - PSNR.

В выданном коде не нужно ничего редактировать. Если ваши функции соответствуют соглашениям, описанным в задании, то априори все должно работать корректно.

Функция `calc_metrics(img_path, c_type, param_list)`. выполняет подсчет PSNR и Rate-Distortion для PCA и JPEG. Также она выполняет построение графиков. В коде не нужно ничего редактировать. На вход функция получает путь до изображения *img_path*; тип сжатия *c_type*; список параметров *param_list*: кол-во компонент в случае PCA, и Quality Factor для JPEG.

Запустите две функции `get_pca_metrics_graph() и get_jpeg_metrics_graph()`. Получившиеся картинки “pca_metrics_graph.png” и “jpeg_metrics_graph.png” сохраните.