Вопрос 1

Верно

Баллов: 1,00 из 1,00

На этом уроке мы с вами реализуем прямой проход сверточного слоя, обратный проход и расчет производных мы трогать не будем.

Вспомним как работает сверточный слой:

- на вход подается массив изображений, еще он называется батчем
- к каждому изображению по границам добавляются нули
- по каждому изображению "скользит" каждый из фильтров сверточного слоя

Давайте начнем с разминки - реализуем функцию, добавляющую padding.

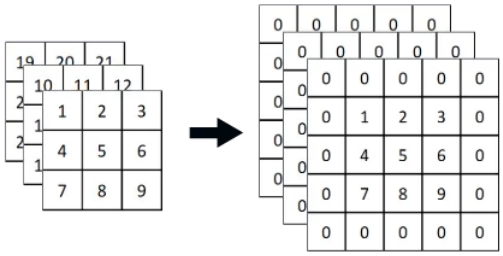
Пусть у нас есть батч input_images из двух изображений с тремя каналами (RGB). Размер изображений пусть будет 3*3. Вспомним, что вход сверточного слоя имеет следующую размерность:

- размер батча
- число каналов
- высота
- ширина

В рассматриваемом случае размерность входа (2, 3, 3, 3).

Если мы добавим вокруг каждого изображения отступ из одного нуля, то размер каждого изображений станет 3+2*1 = 5 пикселей в ширину и 5 в высоту соответственно (добавляем по одному нулю с каждой стороны изображения).

Напишите любую работающую реализацию.



Ответ: (штрафной режим: 0 %)

Сброс ответа

```
import torch

# Создаем входной массив из двух изображений RGB 3*3
input_images = torch.tensor(
      [[[[0,  1,  2],
         [3,  4,  5],
         [6,  7,  8]],

        [[9, 10, 11],
         [12, 13, 14],
         [15, 16, 17]],

        [[18, 19, 20],
         [21, 22, 23],
         [24, 25, 26]]],


       [[[27, 28, 29],
         [30, 31, 32],
         [33, 34, 35]],

        [[36, 37, 38],
         [39, 40, 41],
         [42, 43, 44]],

        [[45, 46, 47],
         [48, 49, 50],
         [51, 52, 53]]]])


def get_padding2d(input_images):
    padded_images = torch.nn.functional.pad(input_images.float(), (1, 1, 1, 1), mode='constant', value=0)
    return padded_images


correct_padded_images = torch.tensor(
      [[[[0.,  0.,  0.,  0.,  0.],
         [0.,  0.,  1.,  2.,  0.],
         [0.,  3.,  4.,  5.,  0.],
         [0.,  6.,  7.,  8.,  0.],
         [0.,  0.,  0.,  0.,  0.]],

        [[0.,  0.,  0.,  0.,  0.],
```

```
          [0.,  9., 10., 11.,  0.],
          [0., 12., 13., 14.,  0.],
          [0., 15., 16., 17.,  0.],
          [0.,  0.,  0.,  0.,  0.]],

         [[0.,  0.,  0.,  0.,  0.],
          [0., 18., 19., 20.,  0.],
          [0., 21., 22., 23.,  0.],
          [0., 24., 25., 26.,  0.],
          [0.,  0.,  0.,  0.,  0.]]],


        [[[0.,  0.,  0.,  0.,  0.],
          [0., 27., 28., 29.,  0.],
          [0., 30., 31., 32.,  0.],
          [0., 33., 34., 35.,  0.],
          [0.,  0.,  0.,  0.,  0.]],

         [[0.,  0.,  0.,  0.,  0.],
          [0., 36., 37., 38.,  0.],
          [0., 39., 40., 41.,  0.],
```

Прошли все тесты! ✔

Верно

Баллы за эту попытку: 1,00/1,00.

Вопрос 2

Верно

Баллов: 1,00 из 1,00

На этом шаге детально рассмотрим из чего состоит сверточный слой.

Сверточный слой это массив фильтров.

Каждый фильтр имеет следующую размерность:

- число слоев во входном изображении (для RGB это 3)
- высота фильтра
- ширина фильтра

В ядре (кернеле) все фильтры имеют одинаковые размерность, поэтому ширину и высоту фильтров называют шириной и высотой ядра. Чаще всего ширина ядра равна высоте ядра, в таком случае их называют размером ядра (kernel_size).

Также слой имеет такие параметры:

- padding - на какое количество пикселей увеличивать входное изображение с каждой стороны.
- stride - на сколько пикселей смещается фильтр при вычислении свертки

Попробуйте самостоятельно вывести формулу размерности выхода сверточного слоя, зная параметры входа и ядра.

Правильность формулы проверьте, сравнив ее с формулой из документации.

Чтобы убедиться в правильности вашей формулы, напишите функцию, принимающую на вход:

- входную размерность (число изображений в батче*число слоев в одном изображении*высота изображения*ширина изображения)
- количество фильтров
- размер фильтров (считаем, что высота совпадает с шириной)
- padding
- stride

Функция должна возвращать размерность выхода.

Ответ: (штрафной режим: 0 %)

Сброс ответа

```
import numpy as np


def calc_out_shape(input_matrix_shape, out_channels, kernel_size, stride, padding):
    batch_size, channels, height, width = input_matrix_shape
    out_height = int((height - kernel_size + 2 * padding) / stride) + 1
    out_width = int((width - kernel_size + 2 * padding) / stride) + 1
    out_shape = [batch_size, out_channels, out_height, out_width]
    return out_shape

print(np.array_equal(
    calc_out_shape(input_matrix_shape=[2, 3, 10, 10],
                   out_channels=10,
                   kernel_size=3,
                   stride=1,
                   padding=0),
    [2, 10, 8, 8]))

# ... и ещё несколько подобных кейсов
```

Прошли все тесты! ✔

Верно

Баллы за эту попытку: 1,00/1,00.

Вопрос **Инфо**

В этом и следующих шагах мы реализуем сверточный слой различными способами.

Тестировать правильность реализации мы будем, сравнивая результаты работы с выходом сверточного слоя из PyTorch.

Для удобства все наши реализации оформим в виде классов. Интерфейс классов сделаем одинаковым и максимально похожим на интерфейс стандартной реализации.

Тестировать наши реализации слоя будем одной и той же функцией.

*Паддить вход вы уже умеете - будем считать, что padding = 0.

В этом шаге вам предлагается изучить код для проверки.

```
import torch
from abc import ABC, abstractmethod


# абстрактный класс для сверточного слоя
class ABCConv2d(ABC):
    def __init__(self, in_channels, out_channels, kernel_size, stride):
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.stride = stride

   def set_kernel(self, kernel):
        self.kernel = kernel

   @abstractmethod
    def __call__(self, input_tensor):
        pass


# класс-обертка над torch.nn.Conv2d для унификации интерфейса
class Conv2d(ABCConv2d):
    def __init__(self, in_channels, out_channels, kernel_size, stride):
        self.conv2d = torch.nn.Conv2d(in_channels, out_channels, kernel_size,
                                      stride, padding=0, bias=False)

   def set_kernel(self, kernel):
        self.conv2d.weight.data = kernel

   def __call__(self, input_tensor):
        return self.conv2d(input_tensor)


# функция, создающая объект класса cls и возвращающая свертку от input_matrix
def create_and_call_conv2d_layer(conv2d_layer_class, stride, kernel, input_matrix):
    out_channels = kernel.shape[0]
    in_channels = kernel.shape[1]
    kernel_size = kernel.shape[2]

   layer = conv2d_layer_class(in_channels, out_channels, kernel_size, stride)
    layer.set_kernel(kernel)

   return layer(input_matrix)


# Функция, тестирующая класс conv2d_cls.
# Возвращает True, если свертка совпадает со сверткой с помощью torch.nn.Conv2d.
def test_conv2d_layer(conv2d_layer_class, batch_size=2,
                      input_height=4, input_width=4, stride=2):
    kernel = torch.tensor(
                      [[[[0., 1, 0],
                         [1,  2, 1],
                         [0,  1, 0]],

                        [[1, 2, 1],
                         [0, 3, 3],
                         [0, 1, 10]],

                        [[10, 11, 12],
                         [13, 14, 15],
                         [16, 17, 18]]]])

   in_channels = kernel.shape[1]

   input_tensor = torch.arange(0, batch_size * in_channels *
                               input_height * input_width,
                               out=torch.FloatTensor()) \
        .reshape(batch_size, in_channels, input_height, input_width)

   custom_conv2d_out = create_and_call_conv2d_layer(
        conv2d_layer_class, stride, kernel, input_tensor)
    conv2d_out = create_and_call_conv2d_layer(
```

```
        Conv2d, stride, kernel, input_tensor)
    return torch.allclose(custom_conv2d_out, conv2d_out) \
                and (custom_conv2d_out.shape == conv2d_out.shape)
print(test_conv2d_layer(Conv2d))
```

Вопрос 3

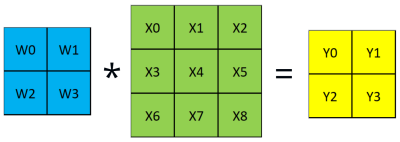
Верно

Баллов: 1,00 из 1,00

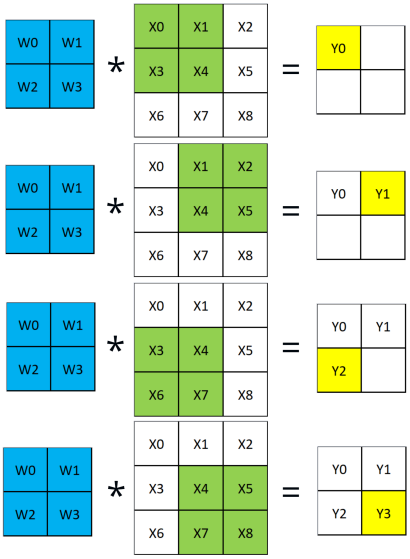
Переиспользуем код с предыдущего шага для проверки своей реализации сверточного слоя.

Рассмотрим свертку батча из одного однослойного изображения 3*3 с ядром из одного фильтра 2*2, stride = 1, то есть, на выходе должна получиться одна матрица 2*2. Строго записанная размерность выхода равна (1 - изображений в батче, 1 - количество фильтров в ядре, 2 - высота матрицы выхода, 2 - ширина матрицы выхода).

Пусть W - веса ядра, X - вход, Y - выход.



Вычислить выход можно в цикле:



На каждой итерации цикла фильтр умножается попиксельно на часть изображения, а потом 4 получившиеся числа складываются - получается один пиксель выхода.

Требуемое количество итераций для данного случая - 4, так как может быть 2 положения ядра и 2 по вертикали, общее число итераций - произведение количеств положений, то есть в данном случае 2*2 = 4.

Давайте перейдем от простого случая к общему.

- **Если бы изображение было многослойным**, например трехслойное - RGB, значит, фильтры в ядре тоже должны быть трехслойные. Каждый слой фильтра попиксельно умножается на соответствующий слой исходного изображения. То есть в данном случае после умножения получилось бы 4*3 = 12 произведений, результаты которых складываются, и получается значение выходного пикселя.
- **Если бы фильтров в ядре было больше одного**, то добавился бы внешний цикл по фильтрам, внутри которого мы считаем свертку для каждого фильтра.
- **Если бы во входном батче было более 1 изображения**, то добавился бы еще один внешний цикл по изображениям в батче.

Напоминание: во всех шагах этого урока мы считаем bias в сверточных слоях нулевым.

На этом шаге требуется реализовать сверточный слой через циклы.

Обратите внимание, что в коде рассматривается общий случай - батч на входе не обязательно состоит из одного изображения, в ядре несколько слоев.

Ответ: (штрафной режим: 0 %)

Сброс ответа

```
    def __init__(self, in_channels, out_channels, kernel_size, stride):
        self.conv2d = torch.nn.Conv2d(in_channels, out_channels, kernel_size,
                                      stride, padding=0, bias=False)

    def set_kernel(self, kernel):
        self.conv2d.weight.data = kernel

    def __call__(self, input_tensor):
        return self.conv2d(input_tensor)


def create_and_call_conv2d_layer(conv2d_layer_class, stride, kernel, input_matrix):
    out_channels = kernel.shape[0]
    in_channels = kernel.shape[1]
    kernel_size = kernel.shape[2]

    layer = conv2d_layer_class(in_channels, out_channels, kernel_size, stride)
    layer.set_kernel(kernel)

    return layer(input_matrix)


def test_conv2d_layer(conv2d_layer_class, batch_size=2,
                      input_height=4, input_width=4, stride=2):
```

```
    kernel = torch.tensor(
                      [[[[0., 1, 0],
                         [1,  2, 1],
                         [0,  1, 0]],

                        [[1, 2, 1],
                         [0, 3, 3],
                         [0, 1, 10]],

                        [[10, 11, 12],
                         [13, 14, 15],
                         [16, 17, 18]]]])

    in_channels = kernel.shape[1]

    input_tensor = torch.arange(0, batch_size * in_channels *
                                input_height * input_width,
                                out=torch.FloatTensor()) \
        .reshape(batch_size, in_channels, input_height, input_width)

    custom_conv2d_out = create_and_call_conv2d_layer(
        conv2d_layer_class, stride, kernel, input_tensor)
    conv2d_out = create_and_call_conv2d_layer(
        Conv2d, stride, kernel, input_tensor)

    return torch.allclose(custom_conv2d_out, conv2d_out) \
             and (custom_conv2d_out.shape == conv2d_out.shape)


# Сверточный слой через циклы.
class Conv2dLoop(ABCConv2d):
    def __call__(self, input_tensor):
        batch_size, _, input_height, input_width = input_tensor.shape
        output_height = (input_height - self.kernel_size) // self.stride + 1
        output_width = (input_width - self.kernel_size) // self.stride + 1
        output_tensor = torch.zeros((batch_size, self.out_channels, output_height, output_width))

        for b in range(batch_size):
            for i in range(0, input_height - self.kernel_size + 1, self.stride):
                for j in range(0, input_width - self.kernel_size + 1, self.stride):
                    for k in range(self.out_channels):
                        output_tensor[b, k, i//self.stride, j//self.stride] = \
                            (self.kernel[k] * input_tensor[b, :, i:i+self.kernel_size, j:j+self.kernel_size]).sum()
        return output_tensor

# Корректность реализации определится в сравнении со стандартным слоем из pytorch.
# Проверка происходит автоматически вызовом следующего кода
# (раскомментируйте для самостоятельной проверки,
#  в коде для сдачи задания должно быть закомментировано):
# print(test_conv2d_layer(Conv2dLoop))
```

Прошли все тесты! ✔

Верно

Баллы за эту попытку: 1,00/1,00.

Вопрос 4

Верно

Баллов: 1,00 из 1,00

Переиспользуем код с третьего шага для проверки своей реализации сверточного слоя.

Реализация через циклы очень неэффективна по производительности. Есть целых два способа сделать то же самое с помощью матричного умножения.

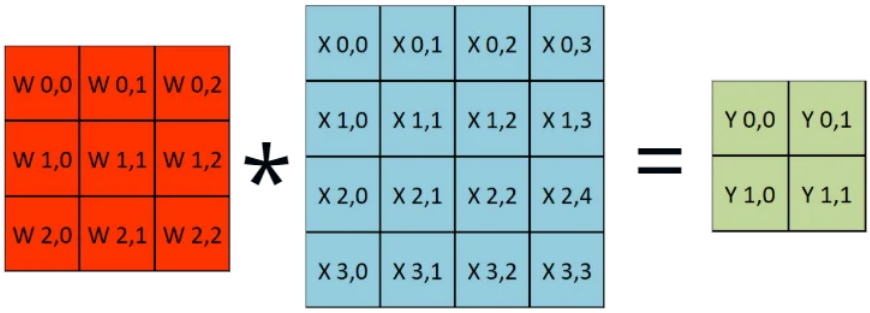
На этом шаге будет реализация первым из них.

Рассмотрим свертку одного одноканального изображения размером 4*4 пикселя (значения пикселей обозначены через X).

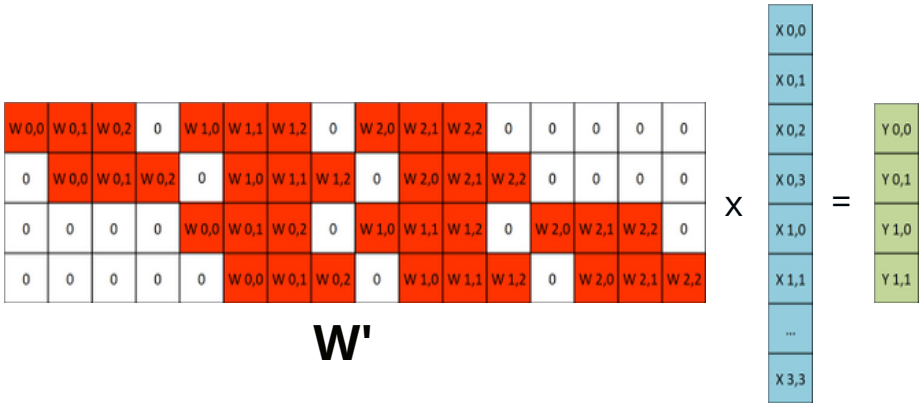
Сворачивать будем с ядром из одного фильтра размером 3*3, веса обозначены через W.

Для простоты примем stride = 1.

Тогда выход Y будет иметь размерность 1*1*2*2 (в данном случае на входе одно изображение - это первая единица в размерности, в ядре один фильтр - это вторая единица в размерности выхода).



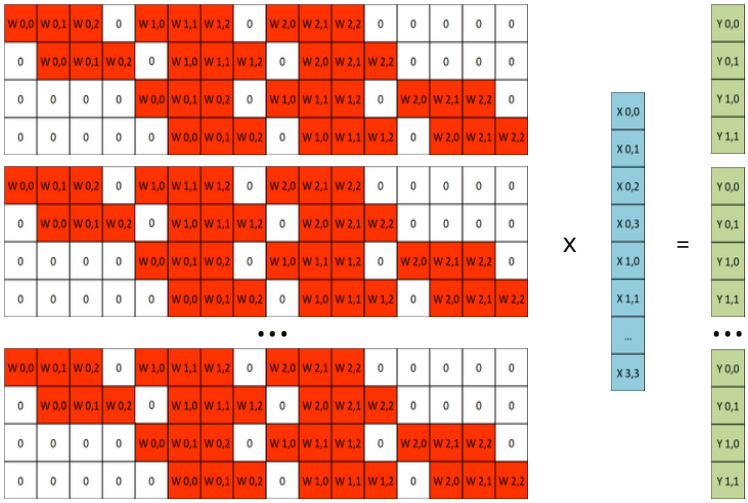
Оказывается, выход свертки можно получить умножением матриц, как показано ниже.



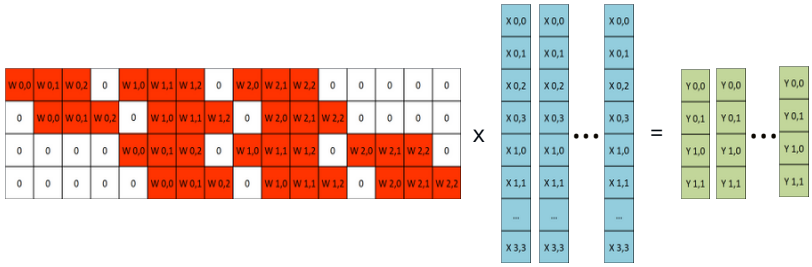
Рекомендуем убедиться в этом, перемножив матрицы на листочке.

Давайте перейдем от простого случая к общему:

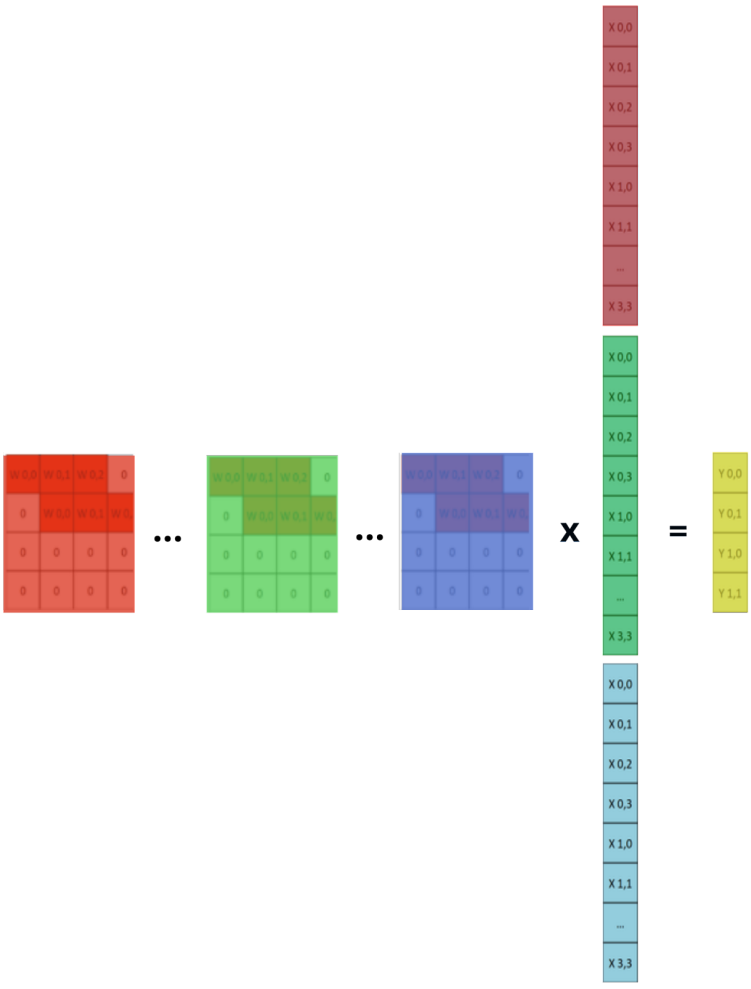
- **Если фильтров в ядре больше одного.** Заметим, что для каждого фильтра, матрица W' будет умножаться на один и тот же вектор изображения. Значит, можно сконкатенировать матрицы фильтров ядра по вертикали и за одно умножение получить ответ для всех фильтров.



- **Если на входе более одного изображения:** заметим, что матрица W' одинакова для всех изображений батча, то есть, можно каждое изображение вначале вытянуть в столбец, а затем эти столбцы для всех изображений батча сконкатенировать по горизонтали.



- **Если в изображении больше одного слоя,** вначале выполним преобразования входа и ядра для каждого слоя, а затем сконкатенируем: вектора разных слоев входа в один большой вектор, а матрицы ядра соответственно в одну длинную матрицу. И мы получим сложение от выходов по слоям в процессе перемножения матриц.



То есть даже в самом общем случае мы за одно умножение матриц можем получить ответ.

Но рассчитанный таким способом выход не совпадает по размерности с выходом стандартного слоя из PyTorch - нужно изменить размерность.

В коде уже реализовано:

- преобразование входного батча изображений
- умножение матрицы ядра на матрицу входа
- преобразование ответа

Напоминание: во всех шагах этого урока мы считаем bias в сверточных слоях нулевым.

Вам осталось реализовать преобразование ядра в описанный выше формат.

Обратите внимание, что в коде рассматривается общий случай - вход состоит из нескольких многослойных изображений, в ядре несколько слоев.

Ответ: (штрафной режим: 0 %)

Сброс ответа

```
import torch
from abc import ABC, abstractmethod

```

```python
def calc_out_shape(input_matrix_shape, out_channels, kernel_size, stride, padding):
    batch_size, channels_count, input_height, input_width = input_matrix_shape
    output_height = (input_height + 2 * padding - (kernel_size - 1) - 1) // stride + 1
    output_width = (input_width + 2 * padding - (kernel_size - 1) - 1) // stride + 1

    return batch_size, out_channels, output_height, output_width


class ABCConv2d(ABC):
    def __init__(self, in_channels, out_channels, kernel_size, stride):
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.stride = stride

    def set_kernel(self, kernel):
        self.kernel = kernel

    @abstractmethod
    def __call__(self, input_tensor):
        pass


class Conv2d(ABCConv2d):
    def __init__(self, in_channels, out_channels, kernel_size, stride):
        self.conv2d = torch.nn.Conv2d(in_channels, out_channels, kernel_size,
                                      stride, padding=0, bias=False)

    def set_kernel(self, kernel):
        self.conv2d.weight.data = kernel

    def __call__(self, input_tensor):
        return self.conv2d(input_tensor)


def create_and_call_conv2d_layer(conv2d_layer_class, stride, kernel, input_matrix):
    out_channels = kernel.shape[0]
    in_channels = kernel.shape[1]
    kernel_size = kernel.shape[2]

    layer = conv2d_layer_class(in_channels, out_channels, kernel_size, stride)
    layer.set_kernel(kernel)

    return layer(input_matrix)


def test_conv2d_layer(conv2d_layer_class, batch_size=2,
```

Прошли все тесты! ✔

Верно

Баллы за эту попытку: 1,00/1,00.