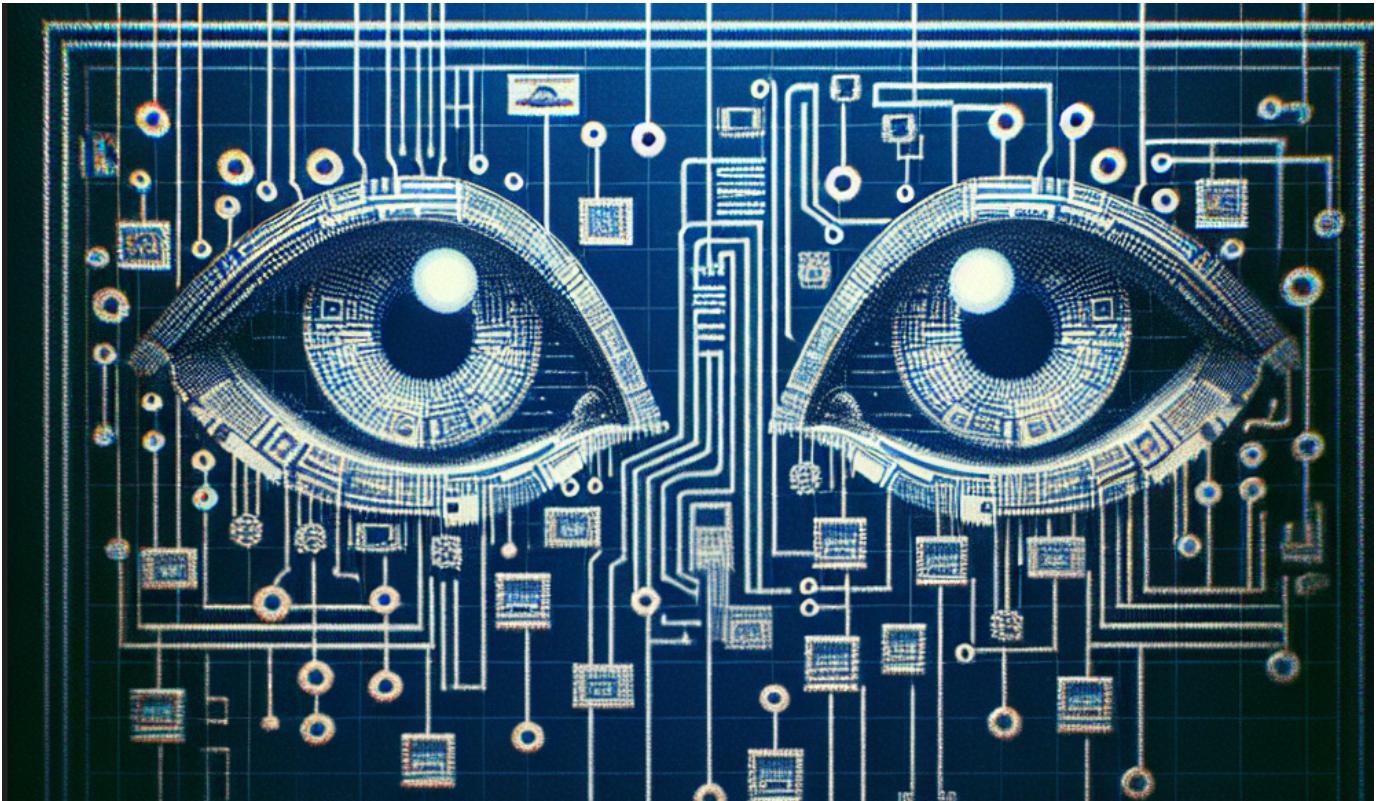


Занятие 13. Методы компьютерного зрения. Свёрточные нейронные сети



Когда речь заходит о по-настоящему интеллектуальных машинах (или вообще о каком-нибудь искусственном разуме в вакууме), представляется, что эти машины способны воспринимать информацию об окружающей среде, и прежде всего, воспринимать посредством механизма зрения. И это не удивляет, поскольку человек через глаза как орган восприятия получает до 80% всей информации.

Компьютерное зрение — это область искусственного интеллекта, занимающаяся разработкой алгоритмов и систем, которые позволяют компьютерам "видеть", анализировать и понимать визуальную информацию из окружающего мира. По сути, задача компьютерного зрения заключается в том, чтобы дать машинам возможность воспринимать изображения и видео так же, как это делает человек.

Основы работы компьютерного зрения

Принципы обработки изображений и извлечение признаков

Почему научить машины видеть так сложно, если человек делает это, не задумываясь? Ответ заключается в том, что восприятие в основном происходит за пределами нашего сознания, внутри специализированных зрительных (и других чувственных модулей) в нашем мозге. К тому времени, когда чувственная информация достигает нашего сознания, она уже оснащена высокогоуровневыми признаками. Например, глядя на фотографию забавного щенка, вы не в состоянии сделать выбор не видеть щенка или не заметить, что он забавный. Вы также не можете объяснить, как вы распознали забавного щенка — для вас же это просто очевидно. Таким образом, мы не можем доверять своему субъективному опыту: восприятие вообще не является тривиальным и для его понимания мы должны посмотреть, как работают чувственные модули.

В отличие от человека, компьютер не сможет идентифицировать объект перед собой без исходных данных и не умеет отделять важное от неважного. Для него цветное изображение похоже на мешочки с цифрами: он «видит» набор пикселей, где каждый пиксель — это три числа, обозначающие количество красного, зелёного и синего по цветовой модели RGB (в случае чёрно-белых изображений — одно число, интенсивность пикселя). Пиксель зелёного цвета может относиться к траве или дому такого же оттенка, но увидеть более полную картину компьютеру не под силу.

Встает вопрос: как "готовить" изображения для обучения компьютера? Рассмотрим основные принципы обработки изображений. Все примеры будут использовать библиотеку OpenCV, которая стала признанным стандартом в области обработки изображений.

In [1]:

```
1 # возьмём картинку в качестве примера
2 import cv2
3 import matplotlib.pyplot as plt
4
5 # чтение изображения
6 image = cv2.imread('data/example.jpg')
7
8 # по-дефолтуopencv читает изображения в формате BGR,
9 # поэтому преобразуем в цветовую схему RGB
10 image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
11
12 # отображение изображения
13 plt.imshow(image)
14 plt.axis('off')
15 plt.show();
```



Что же представляет собой изображение в компьютере вообще и в python для библиотек работы с изображениями в частности? Это знакомый вам массив `numpy.ndarray`.

In [2]:

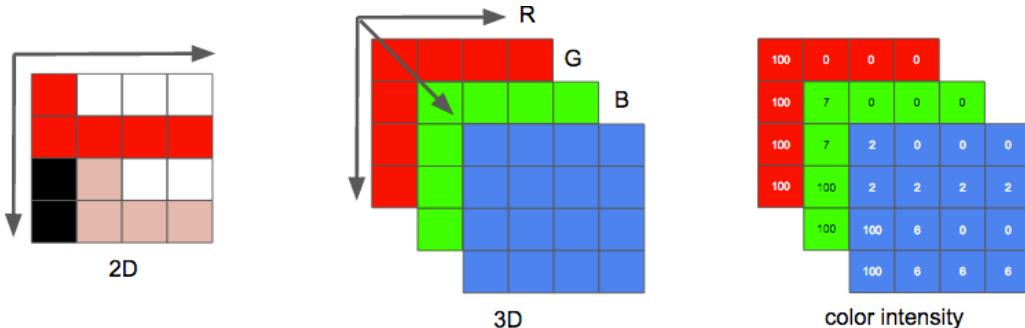
```
1 print(f"Тип данных: {type(image)}")
2 print(f"Количество измерений: {image.ndim}")
3 print(f"Форма массива: {image.shape}")
```

Тип данных: <class 'numpy.ndarray'>

Количество измерений: 3

Форма массива: (506, 700, 3)

Откуда берётся такая форма массива, почему в форме массива присутствует 3? Всё очень просто: массив этого изображения трёхмерный, первые два числа — это ширина и высота изображения (оси x и y), а третье число (третья ось, z) — это количество цветов в кодировке изображения.



Третья ось придаёт объём. Обратите внимание на рисунок выше: по сути, изображение в кодировке RGB (и в любых других кодировках) — это наложение отдельных матриц, или цветовых каналов. В случае кодировки RGB это матрицы красного, зелёного и синего цвета. Мы можем отобразить нашу картинку в оттенках каждого из трёх цветов.

In [3]:

```
1 # массив в исходном виде
2 # каждый подмассив из трёх чисел - отдельный пиксель
3 image
```

Out[3]:

```
array([[[144, 138, 90],
       [144, 138, 90],
       [144, 138, 88],
       ...,
       [ 13,  20,   2],
       [ 14,  21,   5],
       [ 14,  21,   5]],

      [[147, 141, 93],
       [146, 140, 92],
       [145, 139, 89],
       ...,
       [ 13,  20,   2],
       [ 14,  21,   5],
       [ 15,  22,   6]],

      [[152, 144, 97],
       [151, 143, 96],
       [149, 141, 92],
       ...,
       [ 14,  21,   3],
       [ 15,  22,   6],
       [ 16,  23,   7]],

      ...,

      [[ 36,  74,  17],
       [ 36,  74,  17],
       [ 37,  75,  18],
       ...,
       [  6,  30,   8],
       [  6,  30,   8],
       [  6,  30,   8]],

      [[ 33,  71,  14],
       [ 33,  71,  14],
       [ 34,  72,  15],
       ...,
       [  5,  29,   7],
       [  5,  29,   7],
       [  5,  29,   7]],

      [[ 33,  71,  14],
       [ 33,  71,  14],
       [ 34,  72,  15],
       ...,
       [  5,  29,   7],
       [  5,  29,   7],
       [  5,  29,   7]]], dtype=uint8)
```

In [4]:

```
1 import numpy as np
2
3 # разделение каналов
4 r, g, b = cv2.split(image)
5
6 # создаем пустые массивы для остальных каналов
7 zeros = np.zeros(r.shape, dtype=r.dtype)
8
9 # объединяем массивы для отображения отдельных цветовых каналов
10 red_channel = cv2.merge([r, zeros, zeros]) # только красный канал
11 green_channel = cv2.merge([zeros, g, zeros]) # только зеленый канал
12 blue_channel = cv2.merge([zeros, zeros, b]) # только синий канал
13
14 # Отображение отдельных цветовых каналов
15 plt.figure(figsize=(15, 10))
16 plt.subplot(1, 3, 1)
17 plt.imshow(red_channel)
18 plt.axis('off')
19 plt.title('Красный канал')
20
21 plt.subplot(1, 3, 2)
22 plt.imshow(green_channel)
23 plt.axis('off')
24 plt.title('Зелёный канал')
25
26 plt.subplot(1, 3, 3)
27 plt.imshow(blue_channel)
28 plt.axis('off')
29 plt.title('Синий канал')
30
31 plt.show();
```

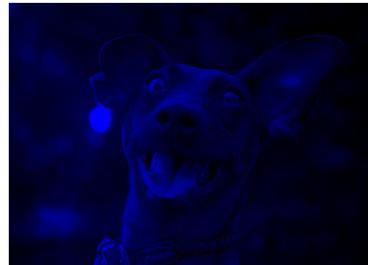
Красный канал



Зелёный канал

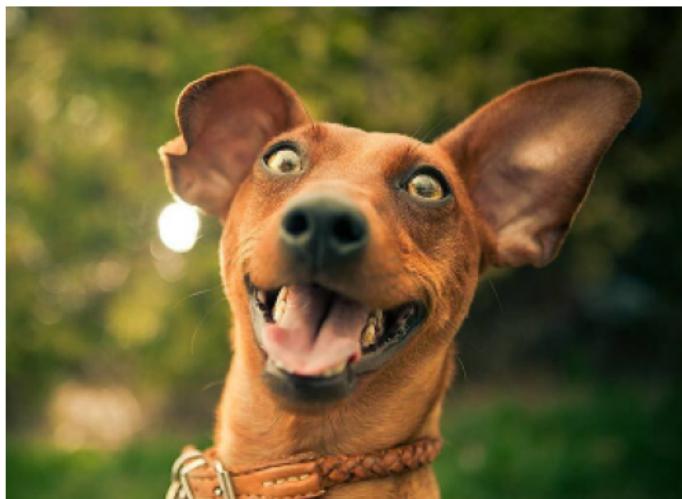


Синий канал



In [5]:

```
1 # можно менять размер изображения
2 img_width, img_height, _ = image.shape
3 plt.imshow(cv2.resize(image, (img_height // 2, img_width // 2)))
4 plt.axis('off')
5 plt.show()
```



По сути, работа с неструктурированными данными (то есть с данными, у которых нет чёткой табличной структуры) — это всегда конструирование и извлечение признаков, даже ещё больше, чем при работе со структуризованными данными.

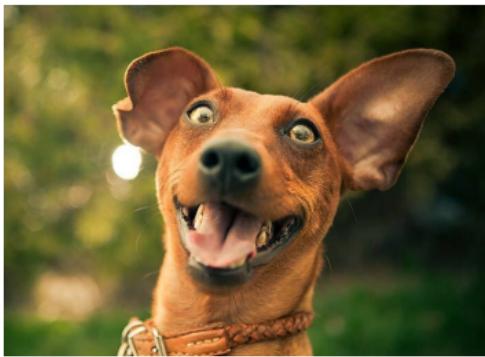
Один из способов преобразовывать картинки для повышения качества — использовать фильтры .

Концепция фильтров особенно знакома тем, кто работал с Photoshop для повышения качества картинки. Просто перечислю некоторые из часто используемых фильтров: размытие, повышение резкости, нахождение краев, тиснение и прочие.

In [6]:

```
1 # добавим гауссовский шум (случайный шум из нормального распределения)
2 mean = 0 # среднее значение гауссовского шума
3 sigma = 25 # стандартное отклонение гауссовского шума
4 gaussian_noise = np.random.normal(mean, sigma, image.shape).astype(np.uint8)
5 noisy_image = cv2.add(image, gaussian_noise)
6
7 plt.figure(figsize=(10, 5))
8 plt.subplot(1, 2, 1)
9 plt.imshow(image)
10 plt.axis('off')
11 plt.title('Исходный вариант')
12
13 plt.subplot(1, 2, 2)
14 plt.imshow(noisy_image)
15 plt.axis('off')
16 plt.title('Вариант с шумом')
17
18 plt.show();
```

Исходный вариант



Вариант с шумом



Как видите, изображение стало менее чётким. Попробуем избавиться от этого "шума" с помощью фильтров размытия.

In [7]:

```
1 # одни из известных фильтров - размытие по Гауссу и медианный фильтр
2 gaussian_image = cv2.GaussianBlur(noisy_image, (7,7),0)
3 median_image = cv2.medianBlur(noisy_image, 3)
4 plt.figure(figsize=(10, 5))
5 plt.subplot(1, 3, 1)
6 plt.imshow(noisy_image)
7 plt.axis('off')
8 plt.title('Зашумлённое изображение')
9
10 plt.subplot(1, 3, 2)
11 plt.imshow(gaussian_image)
12 plt.axis('off')
13 plt.title('С размытие по Гауссу')
14
15 plt.subplot(1, 3, 3)
16 plt.imshow(median_image)
17 plt.axis('off')
18 plt.title('С медианным фильтром')
19
20 plt.show();
```

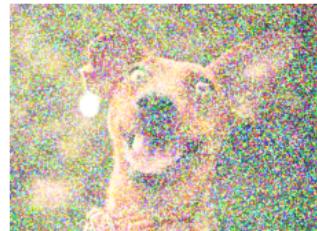
Зашумлённое изображение



С размытие по Гауссу



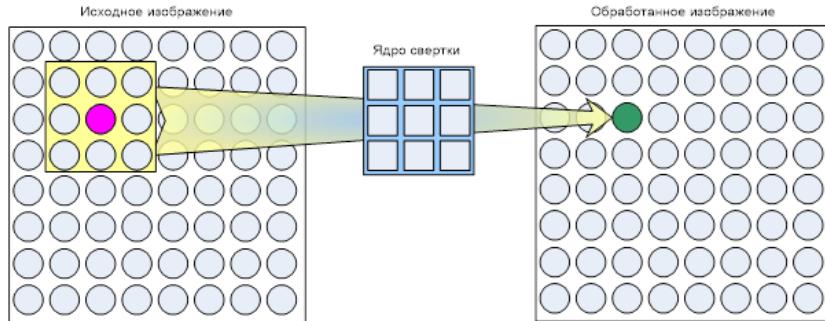
С медианным фильтром



Как видим, с гауссовским шумом лучше всего справилось размытие по Гауссу, но это только в этом случае. Как правило, выбор фильтра \$\$ это эмпирическая задача.

Как же работают эти фильтры? Фильтры, которые были перечислены, а так же множество других основаны на свертке. Свертка (англ. convolution) — это операция, показывающая «схожесть» одной функции с отражённой и сдвинутой копией другой. Понятие свёртки обобщается для функций, определённых на группах, а также мер. Несколько сложноватое определение, не так ли?

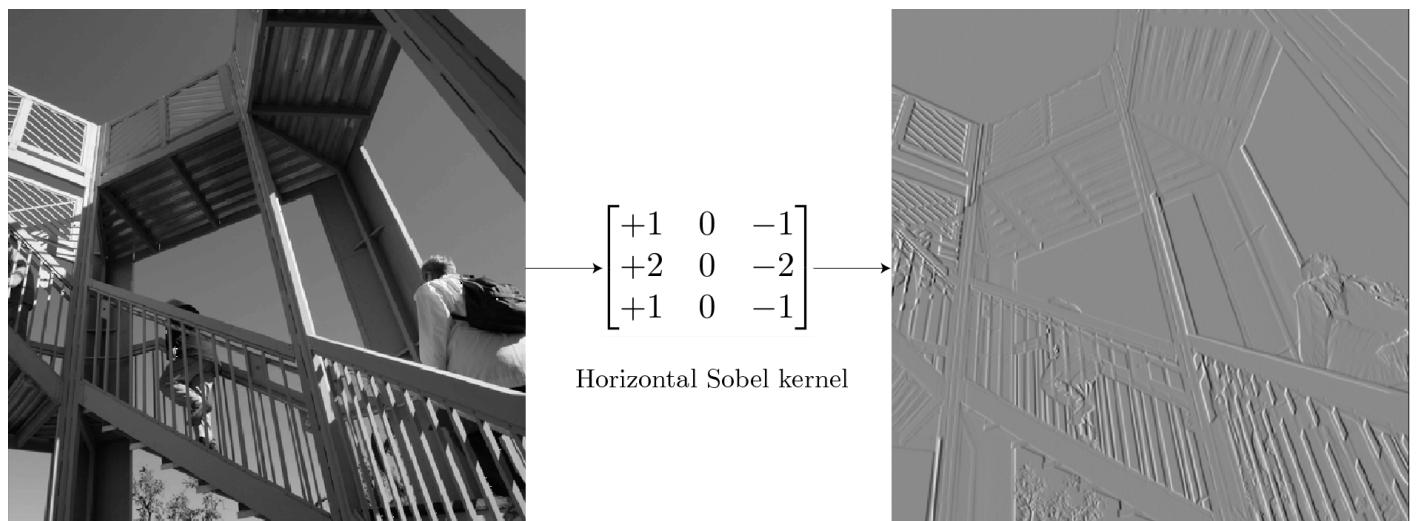
Если говорить проще, то свертка – это операция вычисления нового значения выбранного пикселя, учитывая значения окружающих его пикселей. Для вычисления значения используется матрица, называемая ядром свертки. Обычно ядро свертки является квадратной матрицей $N \times N$, где N — нечетное, однако ничто не мешает сделать матрицу прямоугольной. Во время вычисления нового значения выбранного пикселя ядро свертки как бы «прикладывается» своим центром (именно тут важна нечетность размера матрицы) к данному пикселию. Окружающие пиксели так же накрываются ядром. Далее высчитывается сумма, где слагаемыми являются произведения значений пикселей на значения ячеек ядра, накрывшей данный пиксель. Сумма делится на сумму всех элементов ядра свертки. Полученное значение как раз и является новым значением выбранного пикселя. Если применить свертку к каждому пикселию изображения, то в результате получится некий эффект, зависящий от выбранного ядра свертки.



Вместо суммы можно использовать любую другую функцию. Например, в медианном фильтре среди всех чисел (значений цветовой интенсивности пикселя) выбирается медианное и присваивается выбранному пикслю.

Многие фильтры можно найти в документации к библиотеке `opencv`.

Идея свёртки помогает не только трансформировать исходное изображение, но и помогает выделять новые признаки. Иногда нас могут интересовать только контуры объектов на изображении, а не все его пиксели. Тогда можно попробовать использовать фильтр, основанный на ядре Собеля:



In [8]:

```
1 # или, если к нашему изображению применить
2 sobel_x = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3) # градиент по X
3 sobel_y = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3) # градиент по Y
4
5 # вычисление модуля градиента
6 sobel_magnitude = np.sqrt(sobel_x**2 + sobel_y**2)
7
8 # нормализуем для отображения (в оттенках чёрно-белого, от 0 до 255)
9 sobel_magnitude = cv2.normalize(sobel_magnitude, None, 0, 255, cv2.NORM_MINMAX)
10 sobel_magnitude = np.uint8(sobel_magnitude)
11
12 plt.figure(figsize=(15, 10))
13 plt.subplot(1, 2, 1)
14 plt.imshow(image, cmap='gray')
15 plt.axis('off')
16 plt.title('Исходное изображение')
17
18 plt.subplot(1, 2, 2)
19 plt.imshow(sobel_magnitude, cmap='gray')
20 plt.axis('off')
21 plt.title('После применения фильтра Собеля')
22
23 plt.show();
```

Исходное изображение



После применения фильтра Собеля



Часто используется операция бинаризации изображения.

In [9]:

```
1 # для бинаризации изображение должно быть в оттенках серого
2 gray_image = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
3
4 # применение пороговой обработки для отсеваания пикселей ниже некоторого значения
5 _, binary_image = cv2.threshold(gray_image, 128, 255, cv2.THRESH_BINARY)
6
7 plt.imshow(binary_image, cmap='gray')
8 plt.axis('off')
9 plt.show();
```



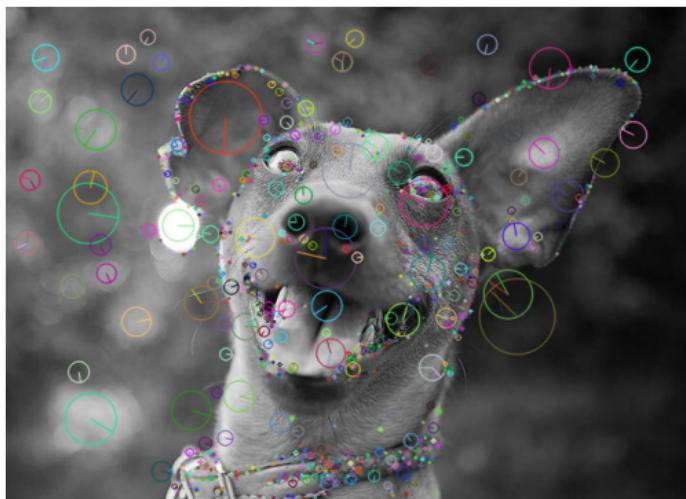
Если вы решаете задачу компьютерного зрения с помощью традиционных методов (моделей семейства регрессии, деревьев и лесов), то можете использовать два алгоритма, которые позволят выделить полезные признаки для них.

SIFT (Scale-Invariant Feature Transform) — это популярный алгоритм для извлечения ключевых точек и описания их признаков в изображениях. Они используются в задачах, таких как распознавание объектов, сшивание изображений и отслеживание движущихся объектов.

In [10]:

```
1 # нас не интересуют цвета, только ключевые точки
2 # поэтому загружаем в цветах серого
3 image = cv2.imread('data/example.jpg', cv2.IMREAD_GRAYSCALE)
4 sift = cv2.SIFT_create()
5
6 # поиск ключевых точек и вычисление описателей
7 keypoints, descriptors = sift.detectAndCompute(image, None)
8
9 # отрисовка ключевых точек на изображении
10 image_with_keypoints = cv2.drawKeypoints(image, keypoints, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
11
12 plt.imshow(image_with_keypoints, cmap='gray')
13 plt.title('SIFT ключевые точки')
14 plt.axis('off')
15 plt.show();
```

SIFT ключевые точки



Задачи компьютерного зрения

Можно выделить четыре основных задачи, для которых используют приёмы компьютерного зрения.

1. **Классификация.** Когда объекту присваивается определённый класс. Простыми словами, машина определяет его в одну из групп, которые она знает: человек, чемодан, скамейка.
2. Задача **локализации** — задача определения точного местоположения объекта на картинке.
3. Задача **обнаружения**. Суть задачи в том, чтобы определить класс объекта на фото и указать его (объекта) местоположение.
4. Задача **сегментации**, которая делится на два вида. Первый — **семантическая сегментация**, которая отделяет изображения от фона и позволяет накладывать на них маски. Ей удастся разделить объекты на классы и выделить масками разного цвета: котов — красной, а собак — зелёной. Другой пример — размытый фон позади человека. На шаг впереди **сегментация объектов**. Котов и собак она распределит по классам, но вдобавок покажет, что они отличаются, и выделит их как разные объекты: собака № 1 и собака № 2.

Классификация	Локализация	Детектирование	Сегментация
			
Кошка	Кошка	Кошка, собака	Кошка, собака

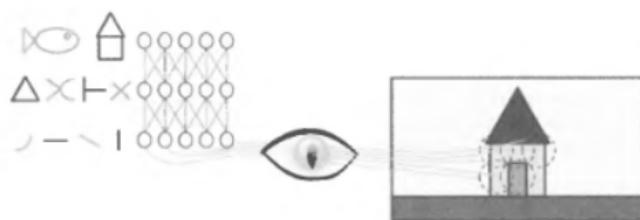
In [11]:

```
1 # пример семантической сегментации
2 # на основе детектора краев Canny
3 edges = cv2.Canny(gray_image, 100, 200)
4
5 plt.imshow(edges, cmap='gray')
6 plt.axis('off')
7 plt.show();
```



Свёрточные нейронные сети

В 1958 и 1959 годах исследователи Дэвид Х. Хьюбел и Торстен Визель провели серию экспериментов на кошках и даже обезьянах и выявили важнейшие сведения о структуре зрительной коры головного мозга. В частности, они показали, что многие нейроны в зрительной коре головного мозга имеют маленькое локальное рецепторное поле, поэтому реагируют только на зрительные раздражители, находящиеся в определённой ограниченной области поля зрения. Поля разных нейронов могут перекрываться и вместе они охватывают все поле зрения.

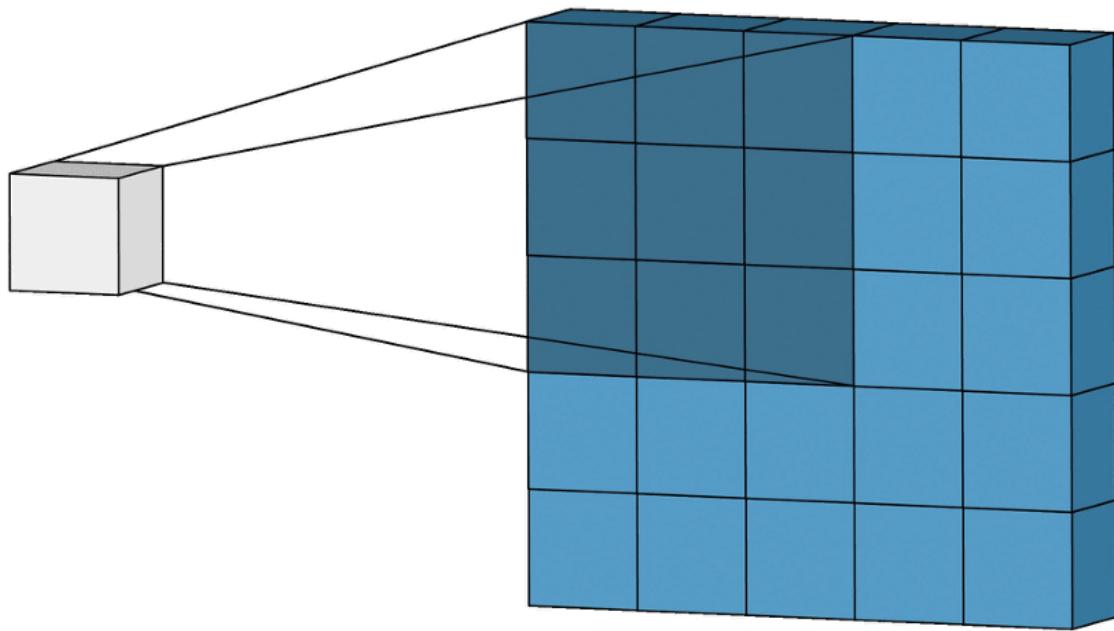


Эти наблюдения привели к мысли, что нейроны более высокого уровня (реагируют на более сложные образы) основываются на выходах соседствующих нейронов более низкого уровня (реагируют на более простые признаки).

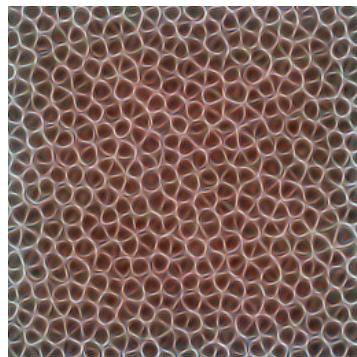
Проведённые исследования зрительной коры вдохновили на создание свёрточных нейронных сетей . Эта архитектура содержит ряд строительных блоков, которые вам уже известны, в том числе полносвязные слои и сигмоидальные функции активации, но она также представляет два новых строительных блока: свёрточные слои (convolutional layer) и объединяющие слои (pooling layer).

Свёрточные слои

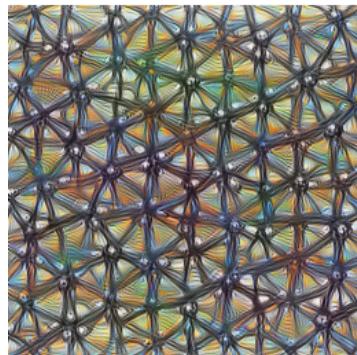
Самый важный строительный блок свёрточной нейронной сети -- это свёрточный слой : нейроны в первом свёрточном слое связаны не с каждым одиночным пикселем, как это было в случае полносвязных персепtronов, а только с пикселями в собственных рецепторных полях. В свою очередь каждый нейрон во втором свёрточном слое связан с нейронами, находящимися внутри небольшого прямоугольника в первом слое.



Такая архитектура позволяет сосредоточиться на маленьких низкоуровневых признаках в первом скрытом слое, а затем скомпоновать их в более крупные высокоуровневые признаки в следующем скрытом слое и т. д.



mixed3a, channel 31



mixed4a, channel 11

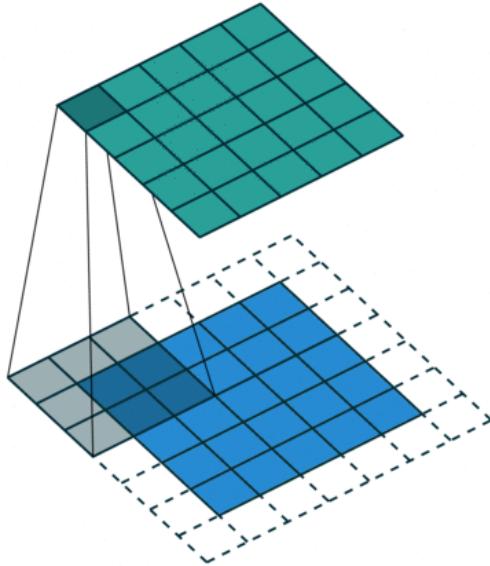


mixed5a, channel 14

Такой подход контрастирует с полно связанными сетями. Так, в приведенном выше примере имеется $5 \times 5 = 25$ входных признаков и $3 \times 3 = 9$ выходных. Если бы это были два полно связанных слоя, весовая матрица состояла бы из $25 \times 9 = 225$ весовых параметров. При этом каждая функция вывода была бы взвешенной суммой всех входов. В случае свертки, взвешенная сумма берется только по числу весов ядра. И в рассмотрении одновременно участвуют только близлежащие элементы.

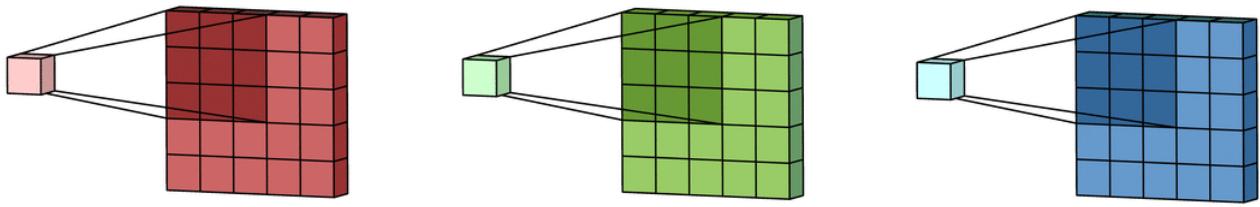
В вышеприведенном примере скольжение ядра «обрезает» исходный двумерный массив по краю, преобразуя матрицу 5×5 в 3×3 . Краевые пиксели теряются из-за того, что ядро не может распространяться за пределы края. Однако иногда необходимо, чтобы размер выходного массива был тем же, что и у входных данных.

Чтобы решить эту задачу, исходный массив можно дополнить «поддельными» пикселями. Например, в виде краевого поля, окружающего массив. Если в качестве значений берутся нули, говорят о «нулевом отступе» (zero padding).

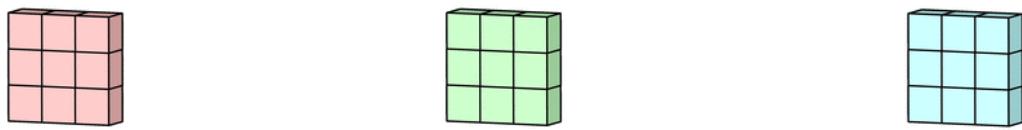


Вышеприведенные диаграммы соответствуют лишь изображениям с одним входным каналом. На практике большинство изображений имеют три канала: красный, зеленый и синий.

В случае с одним каналом термины фильтр и ядро взаимозаменяемы. Для цветного изображения они различны. Фильтр – это коллекция ядер, каждое из которых соответствует одному каналу. Ядро фильтра скользит по данным канала, создавая их обработанную версию. Значимость ядер определяется взаимным отношением их весов. Например, ядро для красного канала может быть более значимым в модели, чем другие ядра фильтра, тогда будут большие и соответствующие веса.

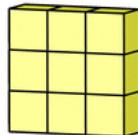


Каждая из обработанных в своих каналах версий суммируется для формирования общего канала.



В выходном терминале может присутствовать линейное смещение, независимое от функций каждого из ядер и свойственное лишь выходному каналу.

Typesetting math: 100%



Как правило, для свёрточных слоёв в библиотеке PyTorch есть класс `torch.nn.Conv2d` (есть и аналогичный ему `torch.nn.Conv1d` для одномерных данных, но для него потребуется "выпрямить" изображение).

In [12]:

```
1 from torch import nn
2
3 convolutional_layer = nn.Conv2d(in_channels=1, # количество входных каналов (1 - для ч/б изображений,
4                                # 3 - для цветных)
5                                out_channels=16, # количество выходных каналов
6                                kernel_size=3, # размер ядра свёртки
7                                stride=1, # шаг окна (сдвинуть окно на 1 пиксель)
8                                padding=1 # дополнение нулями (в кол-ве пикселей от края)
9
10 convolutional_layer
```

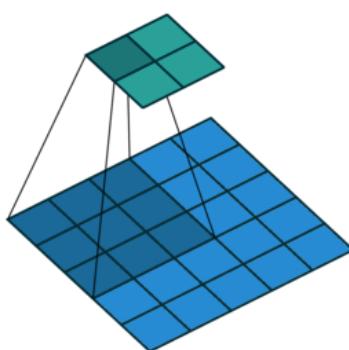
Out[12]:

```
Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

Объединяющие слои

Разобравшись в том, как работают свёрточные слои, понять назначение объединяющих слоёв довольно легко. Их цель заключается в том, чтобы проредить входное изображение для сокращения вычислительной нагрузки, расхода памяти и количества параметров (тем самым сократить риск переобучения).

Как и в свёрточных слоях, каждый нейрон в объединяющем слое связан с выходами ограниченного числа нейронов из предыдущего слоя, которые расположены внутри небольшого прямоугольного рецепторного поля. При этом объединяющий нейрон не имеет весов, он лишь агрегирует входы с применением функции агрегирования: среднее значение, максимальное или минимальное значение и т.д.



In [13]:

```
1 pooling_layer = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
2 pooling_layer
```

Out[13]:

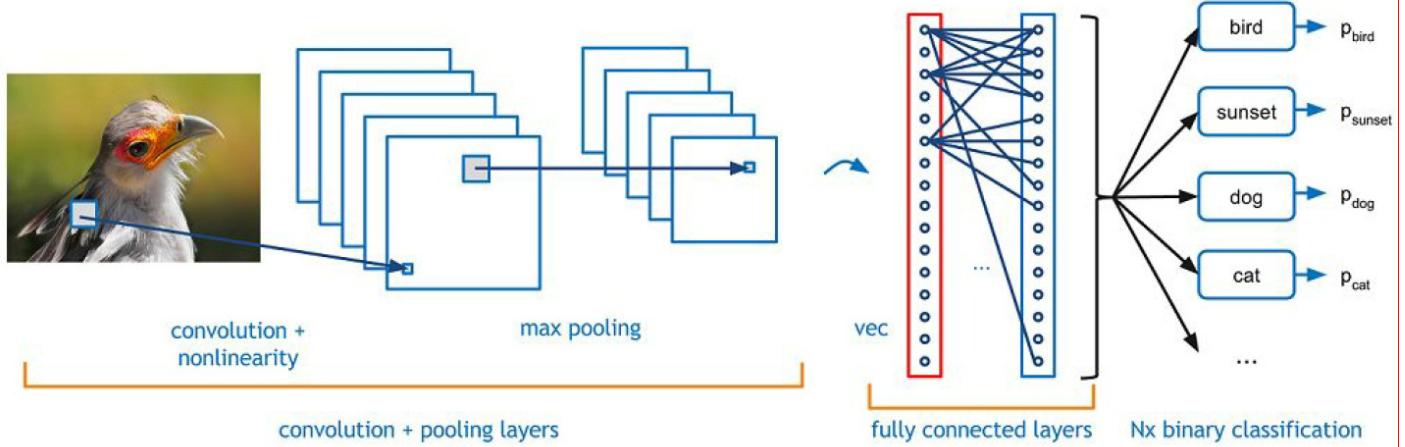
```
MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

Архитектуры свёрточных нейронных сетей

Типовая архитектура свёрточной нейронной сети укладывает стопкой несколько свёрточных слоёв (за каждым из которых следует слой ReLU), объединяющий слой и т. д. По мере прохождения через сеть изображение становится всё меньше и меньше, но обычно также и глубже и глубже. На верхушку добавляется обычная полносвязная сеть и финальный слой выпускает прогноз (в зависимости от задачи).

Свёрточная нейросеть: общий вид

Свёрточная нейросеть (CNN) — это Feed-Forward сеть специального вида:



Простой пример свёрточной сети для распознавания изображений с цифрами из знакомого вам набора данных MNIST мог бы выглядеть следующим образом:

In [14]:

```
1 class ConvNet(nn.Module):
2     def __init__(self):
3         super(ConvNet, self).__init__() # инициализация родительского конструктора
4
5         # слои первого уровня свёртки
6         self.layer1 = nn.Sequential(
7             nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=2),
8             nn.ReLU()
9         )
10
11        # слои второго уровня свёртки
12        self.layer2 = nn.Sequential(
13            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
14            nn.ReLU(),
15            nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
16        )
17
18        self.drop_out = nn.Dropout()
19        self.fc1 = nn.Linear(14 * 14 * 32, 128)
20        self.fc2 = nn.Sequential(
21            nn.Linear(128, 10), # 10 классов, поэтому на выходе 10 нейронов,
22            nn.Softmax(dim=1) # функция активации для многоклассовой классификации
23        )
24
25    def forward(self, x):
26        out = self.layer1(x)
27        out = self.layer2(out)
28        out = out.reshape(out.size(0), -1) # отсеивающий слой принимает выпрямленный вектор
29        out = self.drop_out(out)
30        out = self.fc1(out)
31        out = self.fc2(out)
32
33        return out
```

`nn.Sequential` позволяет объединить несколько слоёв в стопку друг за другом, что позволяет несколько сократить количество полей класса `ConvNet`.

В данном случае мы создаём первый уровень свёрточной сети из двух слоёв — непосредственно свёрточного и активационного (с функцией ReLU). Изображения в наборе данных MNIST чёрно-белые (у них всего один цветовой канал ("256 оттенков серого")), поэтому параметр `in_channels` у первого свёрточного слоя равен 1. Размер ядра свёртки `kernel_size` на первом уровне выбрал достаточно большим, но на последующих уровнях он должен быть меньше. Как правило, свёртка меньшего размера работает эффективнее в плане потребления ресурсов и в плане повышения качества распознавания.

Параметр `padding` (дополнение) устанавливается несколько сложнее. Размер измерения на выходе от операции свёрточной фильтрации или пулинга может быть посчитан с помощью уравнения:

$$W_{\text{out}} = \frac{W_{\text{in}} - \text{размер фильтра} + 2 * \text{размер паддинга}}{\text{страйд}} + 1$$
, где W_{in} — ширина входного изображения, W_{out} — ширина изображения на выходе.

Если фильтр не квадратный (то есть ширина и высота различаются), то по этой формуле можно посчитать и паддинг для высоты.

Если мы хотим оставить входные и выходные измерения без изменений, но с заданными размерами свёртки и шагом (страйдом), то по этой формуле можно найти, что размер отступа должен быть равен 2.

Второй уровень свёртки состоит из трёх слоёв: самого свёрточного, активационного ReLu и объединяющего слоя по максимуму (то есть в качестве агрегирующей функции используется выбор максимального значения среди пикселей в окне).

Здесь нужно обратить внимание на слой подвыборки. Мы будем идти по изображению окном 2x2, делая шаг в 2 пикселя. Это приведёт к тому, что пиксели будут выбираться из непересекающихся областей. Как посчитать, какого размера будет изображение на выходе слоя подвыборки? Просто посчитаем, сколько в квадрате 28x28 (размер изображений) умещается квадратиков 2x2 — 196, а изображение на выходе будет размером 14x14.

После свёрточной части следует просеивающий слой для предотвращения переобучения модели и нейтрализации проблем нестабильных градиентов. Завершается всё скрытым выпрямляющим полно связанным слоем, которому на вход подаётся изображение размера 14x14x32 (32 канала), а на выходе — 128 нейронов (можно менять по своему усмотрению), и выходным полно связанным слоем с 10 нейронами (по числу распознаваемых классов).

Теперь попробуем обучить и оценить эту архитектуру.

In [15]:

```
1 def train_model(model, criterion, optimizer, train_loader, num_epochs):
2
3     total_step = len(train_loader)
4     loss_list = []
5     acc_list = []
6
7     model.train()
8
9     for epoch in range(num_epochs):
10        epoch_acc = []
11        epoch_loss = []
12        for i, (images, labels) in enumerate(train_loader):
13            # прямой проход по сети
14            outputs = model(images)
15            loss = criterion(outputs, labels)
16
17            # обратный проход и оптимизация
18            optimizer.zero_grad()
19            loss.backward()
20            optimizer.step()
21
22            total = labels.size(0)
23            _, predicted = torch.max(outputs.data, 1)
24            correct = (predicted == labels).sum().item()
25
26
27            epoch_acc.append((correct / total) * 100)
28            epoch_loss.append(loss.item())
29
30            avg_loss = np.mean(epoch_loss)
31            avg_acc = np.mean(epoch_acc)
32            loss_list.append(avg_loss)
33            acc_list.append(avg_acc)
34            print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {avg_loss:.4f}, Accuracy: {avg_acc:.2f}%')
35
    return model, loss_list, acc_list
```

In [16]:

```
1 def model_eval(model, test_loader):
2     model.eval()
3     with torch.no_grad(): # отключаем вычисление градиентов (мы не обучаем модель, а оцениваем)
4         correct = 0
5         total = 0
6         for images, labels in test_loader:
7             outputs = model(images)
8             _, predicted = torch.max(outputs.data, 1)
9             total += labels.size(0)
10            correct += (predicted == labels).sum().item()
11
12            print(f'Правильность модели на 10 000 изображений: {(correct / total) * 100 :.2f}%')
13
    return model
```

In [17]:

```
1 import torch
2 import torchvision
3 from torchvision import transforms
4 from torch.utils.data import DataLoader
5
6 DATA_PATH = 'data/'
7
8 LEARNING_RATE = 0.001
9 BATCH_SIZE = 32
10
11 model = ConvNet()
12 criterion = nn.CrossEntropyLoss()
13 optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)
14
15 # загружаем данные
16 trans = transforms.Compose([
17     transforms.ToTensor(), # превращаем в тензор
18     transforms.Normalize((0.1307,), (0.3081,)) # нормализуем картинки
19 ])
20
21 train_dataset = torchvision.datasets.MNIST(root=DATA_PATH, train=True,
22                                              transform=trans, download=True)
23 test_dataset = torchvision.datasets.MNIST(root=DATA_PATH, train=False, transform=trans)
24
25
26 train_loader = DataLoader(dataset=train_dataset, batch_size=BATCH_SIZE, shuffle=True)
27 test_loader = DataLoader(dataset=test_dataset, batch_size=BATCH_SIZE, shuffle=False)
```

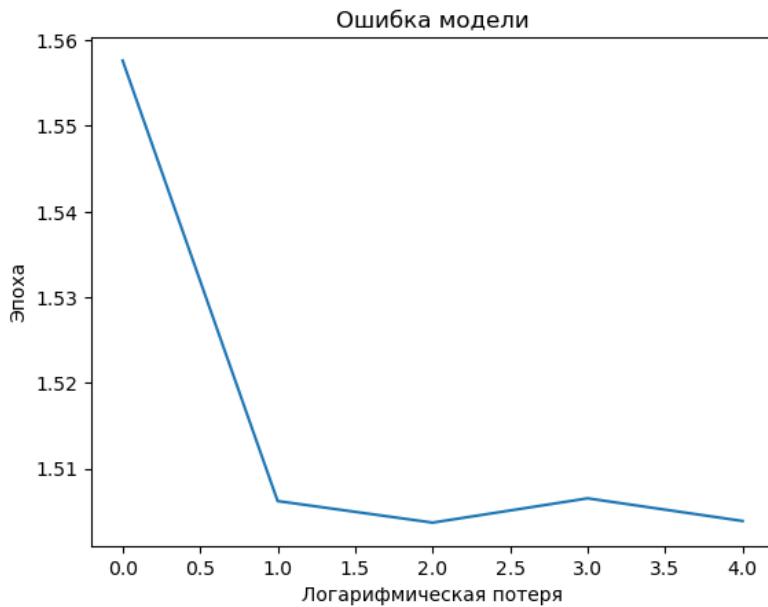
In [18]:

```
1 %%time
2 model, loss_history, acc_history = train_model(model, criterion, optimizer, train_loader, 5)

Epoch [1/5], Loss: 1.5576, Accuracy: 90.39%
Epoch [2/5], Loss: 1.5062, Accuracy: 95.48%
Epoch [3/5], Loss: 1.5037, Accuracy: 95.74%
Epoch [4/5], Loss: 1.5065, Accuracy: 95.44%
Epoch [5/5], Loss: 1.5039, Accuracy: 95.72%
CPU times: user 16min 22s, sys: 14.6 s, total: 16min 37s
Wall time: 2min 5s
```

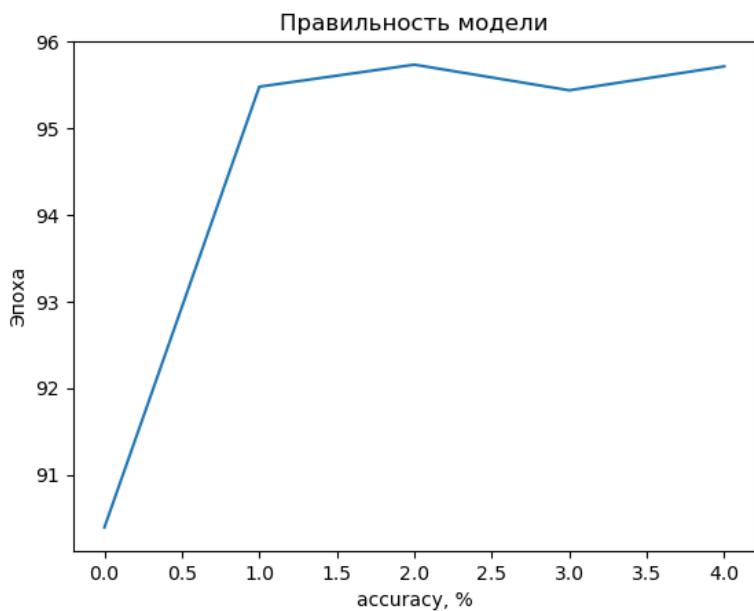
In [19]:

```
1 plt.title("Ошибка модели")
2 plt.xlabel("Логарифмическая потеря")
3 plt.ylabel("Эпоха")
4 plt.plot(loss_history);
```



In [20]:

```
1 plt.title("Правильность модели")
2 plt.xlabel("accuracy, %")
3 plt.ylabel("Эпоха")
4 plt.plot(acc_history);
```



In [21]:

```
1 total_model = model_eval(model, test_loader)
```

Правильность модели на 10 000 изображений: 96.97

Задача обнаружения

Как говорилось в предыдущей лекции, установление местонахождения объекта на фотографии может быть выражено в виде задачи регрессии: распространённый подход к прогнозированию ограничивающей рамки вокруг объекта заключается в том, чтобы прогнозировать горизонтальные и вертикальные координаты центра объекта, а также его ширину и высоту.

Однако есть проблемы. Как правило, наборы данных с изображениями не имеют ограничивающих рамок вокруг объектов. Следовательно, нужно добавлять их самостоятельно. Получение меток часто оказывается одним из самых трудных и дорогостоящих частей проекта по машинному обучению. Имеет смысл потратить какое-то время на поиск подходящих инструментов.

Предположим, что вы получили ограничивающие рамки для всех изображений цветков в наборе данных. Далее понадобится создать набор данных, элементы которого будут пакетами предварительно обработанных изображений наряду с метками классов и их ограничивающими рамками. Каждый элемент должен быть кортежем формы (images, (class_labels, bounding_boxes)). После этого всё готово для обучения модели.

Потеря MSE неплохо работает в качестве функции издержек при обучении модели, но она не является выдающимся показателем для оценки, насколько хорошо модель способна прогнозировать ограничивающие рамки. Самым распространённым показателем для такой цели служит пересечение по объединению: площадь перекрытия между спрогнозированной ограничивающей рамкой и целевой ограничивающей рамкой, делённая на площадь их объединения.



Ещё несколько лет назад обычный подход предусматривал привлечение сверточной сети, обученной классификации и обнаружению местонахождения одиночного объекта, и её плавное продвижение по изображению.

Методика довольно прямолинейна и, как было показано, она будет выявлять тот же самый объект много раз в слегка отличающихся позициях. Затем потребуется заключительная обработка, чтобы избавиться от излишних ограничивающих рамок. Часто используемый подход называется подавлением немаксимумов. Вот в чём он заключается:

- Добавьте к свёрточной сети дополнительный выход объектности для оценки вероятности того, что объект действительно присутствует на изображении. Затем избавьтесь от всех ограничивающих рамок, для которых мера объектности ниже определённого порога: это отбросит все ограничивающие рамки, на самом деле не содержащие нужные объект.
- Найдите ограничивающую рамку с наивысшей мерой объектности и избавьтесь от всех остальных ограничивающих рамок, которые значительно перекрываются с ней.
- Повторяйте шаг 2 до тех пор, пока больше не останется ограничивающих рамок, от которых нужно избавиться.

Другой подход к выявлению объектов предложил в 2015 году Джозеф Редмон. Он представил архитектуру YOLO (You Only Live Once , "живём лишь один раз"). В отличие от описанного выше метода, эта архитектура оказалась очень быстрой. Настолько, что оказалась способной работать в реальном времени.

Эта архитектура нейронной сети выдаёт для каждой ячейки пять ограничивающих рамок, и каждая ограничивающая рамка поступает с мерой объектности. Она также выдает для каждой ячейки решётки 20 вероятностей классов, так как обучалась на наборе данных, который содержит 20 классов. В сумме получается 45 чисел на ячейку решётки: 5 ограничивающих рамок, каждая с 4 координатами, 5 мер объектности и 20 вероятностей классов.

Вместо прогнозирования абсолютных координат центров ограничивающих рамок архитектура YOLO прогнозирует смещение относительно координат ячейки решётки, где $(0, 0)$ левый верхний угол ячейки, а $(1, 1)$ правый нижний угол. Для каждой ячейки решётки YOLO обучается прогнозированию только ограничивающих рамок, чьи центры находятся в этой ячейке (но сама ограничивающая рамка в общем случае может простираться за пределы ячейки решётки). YOLO применяет к координатам ограничивающих рамок логистическую функцию активации, гарантируя, что они остаются в диапазоне 0-1.

Перед обучением нейронной сети YOLO ищет пять репрезентативных измерений ограничивающих рамок, которые называются опорными рамками . Это делается путём применения алгоритма K-Means к высоте и ширине ограничивающих рамок обучающего набора.

Таким образом, получилось с высокой степенью уверенности прогнозировать, что изображение представляет, допустим, собаку, даже если неясно, какой именно породы.

Пример задачи обнаружения

Пример работы с YOLO на torch: <https://www.kaggle.com/code/vencerlanz09/bottle-image-classification-using-yolov5>
[\(https://www.kaggle.com/code/vencerlanz09/bottle-image-classification-using-yolov5\)](https://www.kaggle.com/code/vencerlanz09/bottle-image-classification-using-yolov5)

Как видите, учить и работать с ней требует серьёзных вычислительных ресурсов, но оно того стоит.

Рассмотрим игрушечный пример: будем распознавать белые геометрические фигуры на чёрном фоне. Задача состоит в том, чтобы определить вид фигуры (прямоугольник, треугольник и круг) и её местоположение на картинке.

In [22]:

```
1 n_samples_per_class = 10_000 # на каждый класс по 10 000 изображений
2 image_size = 64 # размер картинки 64x64 пикселя
3 min_size = 12 # отступ фигуры от краёв изображения
```

In [23]:

```
1 def create_rectangleXY(img_size, min_obj_size):
2     x_rect = np.zeros((img_size, img_size, 1), dtype="uint8")
3
4     # задать случайные координаты верхнего левого угла
5     x, y = np.random.randint(0, img_size - min_obj_size, 2)
6
7     # задать случайные размеры ширины и высоты
8     w = np.random.randint(min_obj_size, img_size - x)
9     h = np.random.randint(min_obj_size, img_size - y)
10
11    color = np.random.randint(150, 255, dtype=int)
12    cv2.rectangle(x_rect, (x, y), (x+w,y+h), color, -1, lineType=cv2.LINE_AA)
13
14    return x_rect, [1, 0, 0, x, y, w, h]
```

In [24]:

```
1 def create_circleXY(img_size, min_obj_size):
2     x_circle = np.zeros((img_size, img_size, 1), dtype="uint8")
3
4     radius = np.random.randint(min_obj_size // 2, img_size // 2)
5
6     # центр круга
7     x, y = np.random.randint(radius, img_size - radius, 2)
8
9     color = np.random.randint(150, 255, dtype=int)
10    cv2.circle(x_circle, (x, y), radius, color, -1, lineType=cv2.LINE_AA)
11
12    return x_circle, [0, 1, 0, x-radius, y-radius, radius*2, radius*2]
```

In [25]:

```
1 def create_triangleXY(image_size, min_obj_size):
2     x_triangle = np.zeros((image_size, image_size, 1), dtype="uint8")
3
4     x, y = np.random.randint(0, image_size - min_size, 2)
5
6     w = np.random.randint(min_size, image_size - x)
7     h = np.random.randint(min_size, image_size - y)
8
9     pts = [(x,y), (x+w, y+h), (x+w, y), (x, y+h)]
10
11    pts.pop(np.random.randint(0, len(pts)-1))
12
13    color = np.random.randint(150, 255, dtype=int)
14    cv2.drawContours(x_triangle, [np.array( pts )], 0, color, -1,
15                      lineType=cv2.LINE_AA)
16
17    return x_triangle, [0, 0, 1, x, y, w, h]
```

In [26]:

```
1 np.random.seed(2024)
2
3 X = []
4 Y = []
5 for i in range(n_samples_per_class):
6     # генерируем прямоугольники
7     x_rect, y_rect = create_rectangleXY(image_size, min_size)
8     X.append(x_rect)
9     Y.append(y_rect)
10
11    # генерируем круги
12    x_circle, y_circle = create_circleXY(image_size, min_size)
13    X.append(x_circle)
14    Y.append(y_circle)
15
16    # генерируем треугольники
17    x_triangle, y_triangle = create_triangleXY(image_size, min_size)
18    X.append(x_triangle)
19    Y.append(y_triangle)
20
21 X = np.array( X )
22 Y = np.array( Y )
23
24 print("X: ", X.shape) # (30000, 64, 64, 1)
25 print("Y: ", Y.shape) # (30000, 7)
```

Поскольку размерность для цветовых каналов стоит последней, а для моделей `pytorch` она должна идти сразу после указания количества изображений, есть необходимость поменять форму массива и привести к `(30000, 1, 64, 64)`.

In [27]:

```
1 X_valid = X.transpose(0, 3, 1, 2)
2 X_valid.shape
```

Out[27]:

(30000, 1, 64, 64)

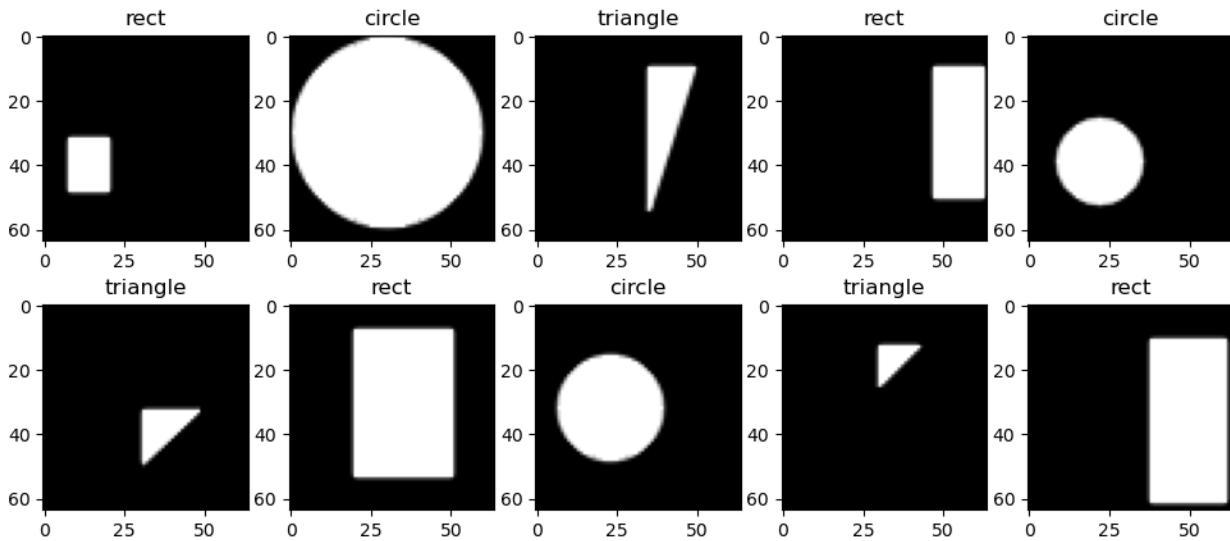
In [28]:

```
1 LABELS = ['rect', 'circle', 'triangle']
2
3 def onehot2label(onehot_vec):
4     return LABELS[np.argmax(onehot_vec)]
```

Посмотрим на получившиеся изображения.

In [29]:

```
1 plt.figure(figsize=(12, 5))
2 for i in range(10):
3     ax = plt.subplot(2, 5, i+1)
4     plt.imshow(X[i], "gray")
5     plt.title(onehot2label(Y[i, :3]) )
6 plt.show()
```



Перемешаем набор данных, чтобы внести случайность, а затем разделим на три поднабора: тренировочный, валидационный (для проверки модели при обучении на каждой эпохе) и тестовый.

In [30]:

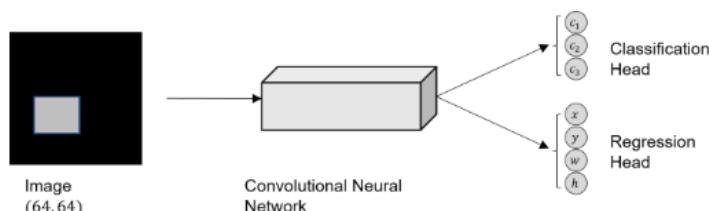
```
1 from sklearn.utils import shuffle
2 X_valid = (X_valid.astype("float32") / 255 ) - 0.5
3 X_valid, Y = shuffle(X_valid, Y)
```

In [31]:

```
1 from sklearn.model_selection import train_test_split
2
3 X_train_val, X_test, Y_train_val, Y_test = train_test_split(X_valid, Y, test_size=0.2)
4 X_train, X_val, Y_train, Y_val = train_test_split(X_train_val, Y_train_val, test_size=0.2, random_state=29)
5
6 print("Shape X train:\t ", X_train.shape) # (19200, 64, 64, 1)
7 print("Shape X validate: ", X_val.shape) # (4800, 64, 64, 1)
8 print("Shape X test:\t ", X_test.shape) # (6000, 64, 64, 1)
```

Shape X train: (19200, 1, 64, 64)
Shape X validate: (4800, 1, 64, 64)
Shape X test: (6000, 1, 64, 64)

Мы опробуем ту же архитектуру для распознавания набора данных MNIST, но внесём изменения.



Как видите, схема показывает, что на вход сети будет подаваться изображение формата $(1, 64, 64)$. В сети представлена свёрточная часть, а полносвязная часть будет иметь два выхода: первый выход для решения задачи регрессии и генерации координат и размеров ограничивающей рамки, второй для решения задачи классификации и предсказания одного из трёх классов фигуры.

In [32]:

```
1 class DetectionNet(nn.Module):
2     def __init__(self):
3         super().__init__()
4
5         # слои первого уровня свёртки
6         self.layer1 = nn.Sequential(
7             nn.Conv2d(1, 16, kernel_size=3, stride=1),
8             nn.ReLU(),
9             nn.MaxPool2d(kernel_size=2, stride=2)
10        )
11
12         # слои второго уровня свёртки
13         self.layer2 = nn.Sequential(
14             nn.Conv2d(16, 32, kernel_size=3, stride=1), # поскольку паддинг был равен 0, мы потеряли немнога в изображении
15                                         # на вход этого уровня пришло изображение размера 30x30
16             nn.ReLU(),
17             nn.MaxPool2d(kernel_size=2, stride=2)
18        )
19
20         self.drop_out = nn.Dropout(p=0.5)
21         self.fc1 = nn.Linear(14 * 14 * 32, 100) # после 2-х операций объединения размер картинки стал 14x14
22         self.output_1 = nn.Linear(100, 4) # этот выходной слой будет выдавать параметры ограничительной рамки
23         self.output_2 = nn.Sequential(
24             nn.Linear(100, 3), # этот выходной слой будет предсказывать один из 3 классов фигуры
25             nn.Softmax(dim=1)
26        )
27
28     def forward(self, x):
29         out = self.layer1(x)
30         out = self.layer2(out)
31         out = out.reshape(out.size(0), -1)
32         out = self.drop_out(out)
33         out = self.fc1(out)
34
35         out_regression = self.output_1(out)
36         out_classif = self.output_2(out)
37         return out_regression, out_classif
```

Для оценки качества локализации объекта, используем уже известную метрику пересечения над объединением.

In [33]:

```
1 import torchvision.ops.boxes as bops
2
3 def calculate_iou(ground, preds):
4
5     def get_xy2(bboxes):
6         # преобразуем (x1, y1, w, h) в (x1, y1, x2, y2)
7         boxes[..., 2] += boxes[..., 0] # x2 = x1 + w
8         boxes[..., 3] += boxes[..., 1] # y2 = y1 + h
9         return boxes
10
11     boxes1 = get_xy2(ground)
12     boxes2 = get_xy2(preds)
13     avg_iou = np.mean((bops.box_iou(boxes1, boxes2)).detach().numpy())
14
15     return avg_iou
```

In [34]:

```
1 def train_detection_model(model, criterion_reg, criterion_classif, optimizer,
2                             train_loader, val_loader, num_epochs):
3
4     total_step = len(train_loader)
5
6     model.train()
7     for epoch in range(num_epochs):
8         epoch_acc, epoch_iou = [], []
9         epoch_mse, epoch_log_loss = [], []
10        for i, (images, answers) in enumerate(train_loader):
11            # прямой проход по сети
12            labels = answers[0][:3]
13            bbox_grounds = answers[0][3:]
14            outputs_reg, outputs_classif = model(images) # будет два вектора с предсказаниями
15            loss_reg, loss_classif = criterion_reg(outputs_reg[0],
16                                                    bbox_grounds), criterion_classif(outputs_classif[0], labels)
17            epoch_log_loss.append(loss_classif.item())
18            epoch_mse.append(loss_reg.item())
19
20            # обратный проход и оптимизация
21            optimizer.zero_grad()
22            loss_reg.backward(retain_graph=True)
23            loss_classif.backward(retain_graph=True)
24            optimizer.step()
25
26            # правильность классификации
27            total = labels.size(0)
28            _, predicted_classes = torch.max(outputs_classif, 1)
29            correct = (predicted_classes == labels).sum().item()
30            epoch_acc.append((correct / total) * 100)
31
32            # пересечение над объединением
33            iou = calculate_iou(outputs_reg, bbox_grounds[None]) * 100
34            epoch_iou.append(iou)
35
36            avg_acc, avg_iou = np.mean(epoch_acc), np.mean(epoch_iou)
37            avg_log_loss, avg_mse = np.mean(epoch_log_loss), np.mean(epoch_mse)
38
39            val_iou, val_acc = compute_metrics(model, val_loader)
40            print(f'Epoch {epoch + 1}/{num_epochs}, log_oss train: {avg_log_loss:.4f}, accuracy train: {avg_acc:.2f}%, i
41            print(f'Epoch {epoch + 1}/{num_epochs}, accuracy val: {val_acc:.2f}%, iou val: {val_iou:.2f}%\n')
42    return model
```

In [35]:

```
1 def compute_metrics(model, loader):
2     model.eval() # режим выработки прогноза
3     iou_list, acc_list = [], []
4     with torch.no_grad():
5         for i, (images, answers) in enumerate(loader):
6             labels = answers[0][:3]
7             bbox_grounds = answers[0][3:]
8             outputs_reg, outputs_classif = model(images)
9             # правильность классификации
10            total = labels.size(0)
11            _, predicted_classes = torch.max(outputs_classif, 1)
12            correct = (predicted_classes == labels).sum().item()
13            acc_list.append((correct / total) * 100)
14
15            # пересечение над объединением
16            iou = calculate_iou(outputs_reg, bbox_grounds[None]) * 100
17            iou_list.append(iou)
18    return np.mean(iou_list), np.mean(acc_list)
```

In [36]:

```
1 from torch.utils.data import TensorDataset, DataLoader
2
3 train_dataset = TensorDataset(torch.Tensor(X_train), torch.Tensor(Y_train))
4 val_dataset = TensorDataset(torch.Tensor(X_val), torch.Tensor(Y_val))
5 test_dataset = TensorDataset(torch.Tensor(X_test), torch.Tensor(Y_test))
6
7 train_loader = DataLoader(train_dataset)
8 val_loader = DataLoader(val_dataset)
9 test_loader = DataLoader(test_dataset)
```

Наконец, обучим модель в течение 35 эпох. Даже нейронную сеть для такого игрушечного примера придётся учить достаточно долго. Для обучения используем GPU.

In [38]:

```
1 %%time
2 det_net = DetectionNet()
3
4 criterion_reg = nn.MSELoss()
5 criterion_classif = nn.CrossEntropyLoss()
6 optimizer = torch.optim.NAdam(det_net.parameters(), lr=LEARNING_RATE)
7
8 model.to('cuda')
9 model = train_detection_model(det_net, criterion_reg, criterion_classif, optimizer,
10                               train_loader, val_loader, 35)
```

Epoch [1/35], log_oss train: 1.0857, accuracy train: 55.53%, iou train: 59.28%
Epoch [1/35], accuracy val: 55.91%, iou val: 69.20%

Epoch [2/35], log_oss train: 0.9602, accuracy train: 42.06%, iou train: 76.25%
Epoch [2/35], accuracy val: 25.74%, iou val: 81.86%

Epoch [3/35], log_oss train: 0.7219, accuracy train: 32.98%, iou train: 80.83%
Epoch [3/35], accuracy val: 31.44%, iou val: 79.46%

Epoch [4/35], log_oss train: 0.6680, accuracy train: 33.23%, iou train: 82.88%
Epoch [4/35], accuracy val: 29.34%, iou val: 84.53%

Epoch [5/35], log_oss train: 0.6483, accuracy train: 33.06%, iou train: 84.02%
Epoch [5/35], accuracy val: 28.75%, iou val: 84.84%

Epoch [6/35], log_oss train: 0.6262, accuracy train: 33.00%, iou train: 84.76%
Epoch [6/35], accuracy val: 36.15%, iou val: 84.82%

Epoch [7/35], log_oss train: 0.6113, accuracy train: 33.19%, iou train: 85.46%
Epoch [7/35], accuracy val: 32.54%, iou val: 86.22%

И в заключение оценим получившуюся модель на отложенном наборе.

In [39]:

```
1 total_model = compute_metrics(model, test_loader)
```

In [40]:

```
1 total_model
```

Out[40]:

```
(88.47177321265141, 33.305555555555555)
```

Как можно заметить, модель научилась достаточно неплохо определять ограничивающую рамку, но распознавать тип фигуры умеет не так хорошо. Вероятно, для выхода классификации можно было бы добавить ещё несколько полносвязных слоёв.

Упражнения

1. Найти информацию о знаменитых архитектурах свёрточных сетей (LeNet, VGGNet и т. д.).
2. Выполнить практическую работу по применению методики обучения с помощью передачи знаний для классификации повреждений сетчатки по результатам ОКТ-сканирования (practice_cv_transfer_learning.ipynb)