

# Конспект по теме «Свёрточные сети»

## Свёртка

Полносвязные сети не могут работать с большими изображениями: если нейронов мало, сеть не найдёт зависимостей, а если много — переобучится. Эту проблему решает свёртка.

Чтобы найти важные для классификации элементы, **свёртка** (англ. *convolution*) ко всем пикселям применяет одинаковые операции.

Разберём, что такое **одномерная свёртка** (англ. *one-dimensional convolution*). Поскольку одномерных изображений не бывает, рассмотрим эту операцию на примере последовательностей.

Пусть  $w$  (англ. *weight*) — это веса,  $s$  (англ. *sequence*) — последовательность.

Свёртка ( $c$ ) выполняется так: веса ( $w$ ) «ползут» вдоль последовательности ( $s$ ), а в каждой позиции вычисляется скалярное произведение. Причём длина вектора весов  $n$  всегда не больше длины вектора последовательности  $m$ . Иначе не будет ни одной позиции, к которой можно применить свёртку.

Запишем операцию одномерной свёртки формулой:

$$c_k = \sum_{t=0}^{n-1} s_{k+t} w_t$$

где  $t$  — индекс для вычисления скалярного произведения,  $k$  — любое значение от 0 до  $(m - n + 1)$ .

Число  $(m - n + 1)$  выбрано так, чтобы веса не выходили за пределы последовательности  $s$ .

Напишем одномерную свёртку на Python:

```
def convolve(sequence, weights):
    convolution = np.zeros(len(sequence) - len(weights) + 1)
    for i in range(convolution.shape[0]):
        convolution[i] = np.sum(weights * sequence[i:i + len(weights)])
```

Теперь посмотрим, как работает **двумерная свёртка** (англ. *two-dimensional convolution, 2D convolution*). Возьмём двумерное изображение  $s$  размером  $m \times m$  пикселей и матрицу весов  $w$  размером  $n \times n$  пикселей. Эту матрицу называют **ядром свёртки** (англ. *kernel*).

Ядро двигается по изображению слева направо и сверху вниз. В каждой позиции её веса умножаются поэлементно на пиксели, полученные произведения суммируются и записываются как пиксели результата.

Для примера свернём такие матрицы:

```
s = [[1, 1, 1, 0, 0],
      [0, 1, 1, 1, 0],
      [0, 0, 1, 1, 1],
      [0, 0, 1, 1, 0],
      [0, 1, 1, 0, 0]]

w = [[1, 0, 1],
      [0, 1, 0],
      [1, 0, 1]]
```

Запишем двумерную свёртку формулой:

$$C_{k1, k2} = \sum_{t_1=0}^n \sum_{t_2=0}^n S_{k1+t1, k2+t2} W_{t1, t2}$$

Операцией свёртки можно найти контуры изображения. Горизонтальные контуры можно найти свёрткой с таким ядром:

```
np.array([[ -1,  -2,  -1],
          [  0,   0,   0],
          [  1,   2,   1]])
```

Это ядро эмпирически вывели американские учёные Ирвин Собель и Гари Фельдман для поиска контуров изображения. Оно выделяет контуры лучше, чем рассмотренное в начале урока ядро.

Чтобы найти вертикальные контуры, применим такое ядро:

```
np.array([[ -1,  0,  1],  
         [ -2,  0,  2],  
         [ -1,  0,  1]])
```

## Свёрточный слой

Разберём, что такое свёрточные слои и какую они роль играют в **свёрточных нейросетях** (англ. *convolutional neural network, CNN*).

**Свёрточные слои** (англ. *convolution layers*) применяют операцию свёртки к входным изображениям. Они выполняют большую часть вычислительной работы сети.

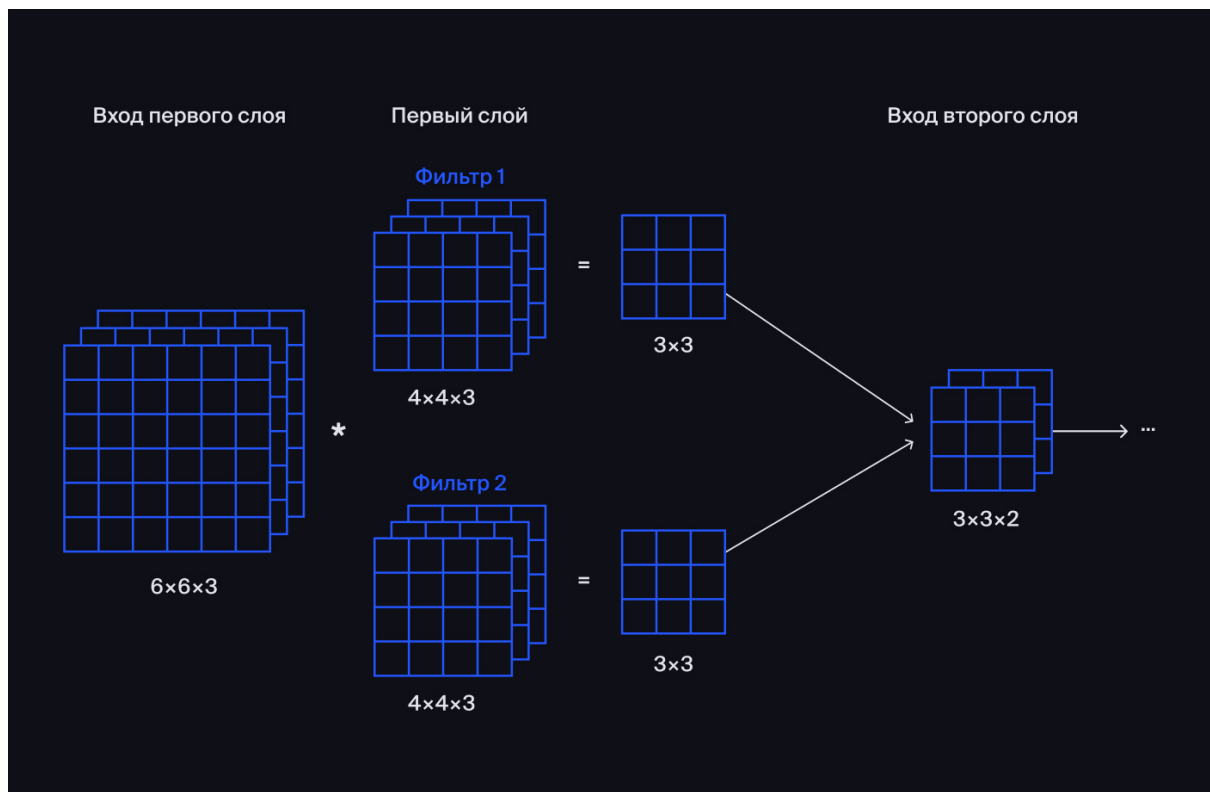
Свёрточный слой состоит из настраиваемых и обучаемых **фильтров** (англ. *filter*) — наборов весов, которые применяются к изображению. По сути это квадратная матрица размером  $K \times K$  (от слова *kernel*) пикселей.

Если изображение цветное, к фильтру добавляется ещё и **глубина** (англ. *depth*), то есть третье измерение. И тогда фильтр уже не матрица, а тензор, или многомерная таблица.

Рассмотрим пример. Перед вами три канала: красный, синий, зелёный. Фильтр размером  $3 \times 3 \times 3$  (три пикселя по ширине, высоте и глубине) последовательно скользит по входному изображению в каждом канале, выполняя операцию свёртки. По третьему измерению он не двигается. Для разных цветов веса разные. Полученные изображения поэлементно складываются, образуя результат свёртки.

Фильтров в свёрточном слое может быть несколько. Каждый фильтр возвращает двумерное изображение, из которого можно снова сделать трёхмерное. На следующем свёрточном слое глубина фильтров равна числу фильтров на предыдущем слое.

Знак звёздочки (\*) означает операцию свёртки.



Параметров в свёрточных слоях намного меньше, чем в полносвязных. Значит, свёрточные слои легче обучать.

Рассмотрим настройки свёрточного слоя:

1. **Padding** (англ. «отбивка»). Эта техника добавляет к краям матрицы нули (англ. *zero padding*), чтобы крайние пиксели участвовали в свёртке не меньше раз, чем центральные. Так важная информация в изображении потеряна не будет. Добавленные нули также участвуют в свёртке. Величина паддинга задаёт толщину отбивки из нулей.



2. **Striding**, или **Stride** (англ. «большой шаг»). Эта техника сдвигает фильтр не на один пиксель, а на большее количество пикселей. Она применяется, когда нужно получить выходное изображение меньшего размера.

Вычислим размер выходного тензора после свёрточного слоя. Если у начального изображения размером  $W \times W \times D$  (от слов *depth*) есть фильтр  $K \times K \times D$ , паддинг ( $P$ ) и шаг ( $S$ ), то новый размер изображения  $W'$  вычисляется по формуле:

$$W' = \frac{(W - K + 2P)}{S} + 1$$

Чтобы лучше усвоить, как работают свёртки с разными параметрами, попробуйте [этот визуализатор на GitHub](#).

## Свёрточные слои Keras

Создадим свёрточный слой **Conv2D** (от англ. *two-dimensional convolution*):

```
keras.layers.Conv2D(filters, kernel_size, strides, padding, activation)
```

Разберём все параметры:

- **filters** — количество фильтров, которому равна величина выходного тензора.
- **kernel\_size** (англ. «размер ядра свёртки») — пространственный размер фильтра  $K$ . Повторим, фильтр — это тензор размером  $K \times K \times D$ , где  $D$  равна глубине входного изображения.
- **strides** — размер шага свёртки. По умолчанию величина *strides* равна единице.
- **padding** — толщина отбивки из нулей. Есть два типа паддинга: *valid* (англ. «допустимый») и *same* (англ. «одинаковый»). Тип паддинга по умолчанию — *valid*, то есть нулевой. Тип *same* означает автоматический подбор величины паддинга так, чтобы ширина и высота выходного тензора была равна ширине и высоте входного тензора.
- **activation** — активация, которую применяют сразу же после свёртки с фильтром. Можно указать знакомые вам функции активации: `'relu'` и `'sigmoid'`. По умолчанию этот параметр равен *None*, то есть активация отключена.

Чтобы результаты свёрточного слоя можно было передать полносвязному слою, подключим новый слой **Flatten** (англ. «разглаживать»). Он делает многомерный тензор одномерным.

Например, после входного изображения размером  $32 \times 32 \times 3$  идёт свёрточный слой, а потом — полносвязный. Между ними нужно вставить слой *Flatten*:

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D, Flatten, Dense

model = Sequential()

# здесь у тензора размер (None, 32, 32, 3)
# первая размерность отвечает за разные объекты
# она равна None, потому что размер батча ещё не известен

model.add(Conv2D(filters=4, kernel_size=(3, 3), input_shape=(32, 32, 3)))

# здесь у тензора размер (None, 30, 30, 4)
```

```
model.add(Flatten())

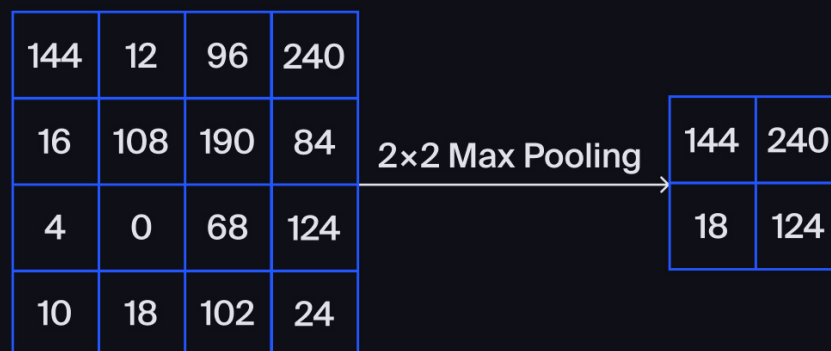
# здесь у тензора размер (None, 3600)
# где 3600 = 30 * 30 * 4

model.add(Dense(...))
```

## Архитектура LeNet

Уменьшить количество параметров модели можно техникой **пулинга** (англ. *pooling*, «объединение»). Технику рассмотрим на примере **Max Pooling** (англ. «максимальный пулинг»):

1. Выбирается размер ядра, например, 2x2;
2. Ядро начинает двигаться слева направо и сверху вниз, в каждом окне из четырёх пикселей находится пиксель с максимальным значением;
3. Пиксель с максимальным значением остаётся, его ближайшие соседи исчезают;
4. Получаем матрицу, состоящую только из пикселей с максимальными значениями.



В Keras можно выполнить не только операцию *MaxPooling*, но и **AveragePooling** (англ. «средний пулинг»). Разберём, в чём их разница:

- *MaxPooling* возвращает максимальное внутри канала значение пикселя из группы. Если у входного изображения размер  $W \times W$ , то у выходного изображения размер будет  $W/K$ , где  $K$  — размер ядра.
- *AveragePooling* возвращает среднее значение из группы пикселей внутри канала.

Рассмотрим запись операции *AveragePooling* в *Keras*:

```
keras.layers.AveragePooling2D(pool_size=(2, 2), strides=None, padding='valid', ...)
```

Разберём каждый параметр:

- **pool\_size** — размер пулинга: чем он больше, тем шире охват задействованных соседних пикселей.
- **strides** — величина шага. Если указано *None*, то шаг равен размеру пулинга.
- **padding** — толщина отбивки из нулей. Тип паддинга *valid* означает нулевой (по умолчанию), тип *same* — автоматический подбор величины паддинга.

Параметры *MaxPooling2D* аналогичны этим параметрам.

Теперь инструментов достаточно, чтобы построить популярную архитектуру для классификации изображений размером 20–30 пикселей — **LeNet**.

Название сети произошло от имени её создателя Яна Лекуна (*Yann André LeCun*), разработчика технологии сжатия изображений *DjVu*, главы лаборатории искусственного интеллекта *Facebook*.

*LeNet* строится так:

1. Сеть начинается с 2–3 свёрточных слоёв размером  $5 \times 5$ , чередующихся с *Average Pooling* размером  $2 \times 2$ . Постепенно они уменьшают пространственное разрешение и собирают разбросанную по всему изображению информацию в матрицы маленького размера (около 5 пикселей).
2. Чтобы не потерять важную информацию, количество фильтров растёт от слоя к слою.



3. В конце сети идёт 1–2 полносвязных слоя, которые собирают все признаки и классифицируют их.

Реализация LeNet на Keras:

```
model = Sequential()

model.add(Conv2D(6, (5, 5), padding='same', activation='tanh',
                input_shape=(28, 28, 1)))
model.add(AvgPool2D(pool_size=(2, 2)))

model.add(Conv2D(16, (5, 5), padding='valid', activation='tanh'))
model.add(AvgPool2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(120, activation='tanh'))

model.add(Dense(84, activation='tanh'))

model.add(Dense(10, activation='softmax'))

model.compile(loss='sparse_categorical_crossentropy', optimizer='sgd', metrics=['acc'])
model.summary()
```

## Алгоритм Adam

Градиентный спуск (*SGD*) — это не самый оптимальный алгоритм обучения нейронной сети. Если величина шага слишком маленькая, сеть будет обучаться долго, а если большая — может пропустить минимум. Чтобы подбор шага был автоматическим, применяют алгоритм **Adam** (от англ. *adaptive moment estimation*, «адаптивность на основе оценки моментов»). Он подбирает различные параметры для разных нейронов, что также ускоряет обучение модели.

Чтобы понять, как этот алгоритм работает, рассмотрим визуализацию Эмильена Дюпона из Оксфордского университета. В ней четыре алгоритма: слева *SGD*, справа *Adam*, а между ними — два похожих на *Adam* алгоритма (их разбирать не будем). Быстрее всех минимум находит *Adam*.

Запишем алгоритм *Adam* в *Keras*:

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
              metrics=['acc'])
```

Чтобы настроить гиперпараметры, подключим класс алгоритма:

```
from tensorflow.keras.optimizers import Adam
optimizer = Adam()

model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy',
              metrics=['acc'])
```

Основной настраиваемый гиперпараметр в алгоритме *Adam* — скорость обучения (*learning rate*). Это шаг градиентного спуска, с которого алгоритм стартует. Записывается так:

```
optimizer = Adam(lr=0.01)
```

По умолчанию он равен 0.001. Уменьшение шага иногда может замедлить обучение, но улучшить итоговое качество модели.

## Загрузчики данных

Массивы хранятся не на жёстком диске компьютера, а только в оперативной памяти. Представьте, нужно сделать массив из терабайта изображений. Даже представить такое сложно, не то что выполнить... Ресурсы оперативной памяти не безграничны!

Чтобы справиться с таким объёмом изображений, к работе подключают динамическую загрузку данных.

В библиотеке *Keras* есть удобный загрузчик **ImageDataGenerator** (англ. «генератор данных изображений»):

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

Из фотографий в папках класс *ImageDataGenerator* формирует батчи с изображениями и метками классов. Создадим его:

```
# от англ. data generator
datagen = ImageDataGenerator()
```

Чтобы загрузчик извлёк данные из папки, вызовем функцию **flow\_from\_directory()** (англ. «поток из директории»):

```
datagen_flow = datagen.flow_from_directory(
    # папка, в которой хранится датасет
    '/dataset/',
    # к какому размеру приводить изображения
    target_size=(150, 150),
    # размер батча
    batch_size=16,
    # в каком виде выдавать метки классов
    class_mode='sparse',
    # фиксируем генератор случайных чисел (от англ. random seed)
    seed=12345)
```

```
Found 1683 images belonging to 12 classes.
```

Загрузчик нашёл 12 классов (папок), всего в них 1683 изображений.

Поясним некоторые аргументы:

- `target_size=(150, 150)` — аргумент с шириной и высотой, к которым будут приводиться изображения. В папках могут лежать изображения разного размера, а нейронным сетям нужно, чтобы все изображения были одинаковые.
- `batch_size=16` — количество изображений в батче. Чем больше изображений, тем лучше обучится сеть. Много фотографий в памяти GPU не поместится, поэтому 16 — это золотая середина, с которой можно стартовать.
- `class_mode='sparse'` — аргумент, который указывает тип выдачи метки классов. `sparse` (англ. «редкий») означает, что метки будут порядковым номером папки.

Узнать, как номера классов связаны с названиями папок, можно так:

```
# англ. индексы классов
print(datagen_flow.class_indices)
```

Применив метод `datagen.flow_from_directory(...)`, получим объект, у которого пары «картинки — метки» можно получить функцией `next()` (англ.

«следующий»):

```
features, target = next(datagen_flow)

print(features.shape)
```

Получили признаки — четырёхмерный тензор, в котором 16 изображений размером 150×150 с тремя цветовыми каналами.

Чтобы обучить на этих данных модель, передадим объект `datagen_flow` в метод `fit()`. Эпоха не должна быть бесконечно длинной. Для этого укажем в параметре `steps_per_epoch` количество батчей в наборе данных:

```
model.fit(datagen_flow, steps_per_epoch=len(datagen_flow))
```

В методе `fit()` должна быть не только обучающая, но и валидационная выборка. Для этого создайте два загрузчика: для обучающей и валидационной.

```
# указываем загрузчику, что валидация содержит
# 25% случайных объектов
datagen = ImageDataGenerator(validation_split=0.25)

train_datagen_flow = datagen.flow_from_directory(
    '/datasets/fruits_small/',
    target_size=(150, 150),
    batch_size=16,
    class_mode='sparse',
    # указываем, что это загрузчик для обучающей выборки
    subset='training',
    seed=12345)

val_datagen_flow = datagen.flow_from_directory(
    '/datasets/fruits_small/',
    target_size=(150, 150),
    batch_size=16,
    class_mode='sparse',
    # указываем, что это загрузчик для валидационной выборки
    subset='validation',
    seed=12345)
```

Теперь обучение запускается так:

```
model.fit(train_datagen_flow,  
          validation_data=val_datagen_flow,  
          steps_per_epoch=len(train_datagen_flow),  
          validation_steps=len(val_datagen_flow))
```

## Аугментации

Если обучающих примеров мало, сеть может переобучиться. Чтобы увеличить датасет, применяют аугментацию. **Аугментация** (англ. *augmentation*) — увеличение объёма данных через трансформацию существующего датасета. Причём меняется только обучающая выборка, тестовая и валидационная остаются прежними.

Суть техники заключается в преобразовании исходного изображения, но с сохранением его целевого признака. Например, изображение можно повернуть или отразить.

Аугментации бывают такие:

- поворот,
- отражение,
- изменение яркости и контрастности,
- растягивание и сжатие,
- размытие и повышение чёткости,
- добавление шума.

К одному изображению можно применить сразу несколько типов аугментаций.

При аугментации можно столкнуться с рядом проблем. Например, поменяется класс изображения или получится не фотография, а картина в стиле абстрактного импрессионизма. От этого пострадает качество модели.

Проблем можно избежать, если следовать этим рекомендациям:

- Не применяйте аугментацию на валидационной и тестовой выборках, чтобы не исказить значения метрик.

- Не запускайте сразу все аугментации: добавляйте по одной и следите за изменением метрики качества на валидационном наборе.
- Всегда оставляйте часть изображений в данных без изменений.

## Аугментации в Keras

В *ImageDataGenerator* есть опции для добавления аугментаций. По умолчанию они отключены. Допустим, применим отражения по вертикали (англ. *vertical flip*):

```
datagen = ImageDataGenerator(validation_split=0.25,  
                             rescale=1./255,  
                             vertical_flip=True)
```

Для обучающей и валидационной выборок нужно создать разные генераторы:

```
train_datagen = ImageDataGenerator(  
    validation_split=0.25,  
    rescale=1./255,  
    horizontal_flip=True)  
  
validation_datagen = ImageDataGenerator(  
    validation_split=0.25,  
    rescale=1./255)  
  
train_datagen_flow = train_datagen.flow_from_directory(  
    '/dataset/',  
    target_size=(150, 150),  
    batch_size=16,  
    class_mode='sparse',  
    subset='training',  
    seed=12345)  
  
val_datagen_flow = validation_datagen.flow_from_directory(  
    '/dataset/',  
    target_size=(150, 150),  
    batch_size=16,  
    class_mode='sparse',  
    subset='validation',  
    seed=12345)
```

Чтобы обучающая и валидационная выборки не содержали общих элементов, задайте объектам *train\_datagen\_flow* и *val\_datagen\_flow*

одинаковое значение `seed`.

## ResNet в Keras

Импортируем *ResNet* из *Keras*. 50 означает количество слоёв в сети.

```
from tensorflow.keras.applications.resnet import ResNet50

model = ResNet50(input_shape=None,
                  classes=1000,
                  include_top=True,
                  weights='imagenet')
```

Рассмотрим все аргументы:

- `input_shape` — размер входного изображения. Например: `(640, 480, 3)`.
- `classes=1000` — количество нейронов в последнем полносвязном слое, в котором выполняется классификация.
- `weights='imagenet'` (от англ. «сеть изображений») — инициализация весов. *ImageNet* — название большого датасета, на котором сеть обучалась классифицировать изображения на 1000 классов. Если обучение сети начать на *ImageNet*, а продолжить на вашей задаче, результат будет лучше, чем если обучать с нуля. Чтобы инициализация весов была случайной, напишите `weights=None`.
- `include_top=True` (англ. «добавить верхушку») — указание на то, что в конце архитектуры *ResNet* есть два слоя: *GlobalAveragePooling2D* и *Dense*. Если задать *False*, то этих слоёв не будет.

Рассмотрим последние слои:

1. *GlobalAveragePooling2D* (англ. «глобальный двумерный пулинг усреднением») — пулинг с окном во весь тензор. Как и *AveragePooling2D*, возвращает среднее значение из группы пикселей внутри канала. *GlobalAveragePooling2D* нужен, чтобы усреднить информацию по всему изображению, то есть получить пиксель с большим количеством каналов (например, 512 для *ResNet50*).
2. *Dense* — полносвязный слой для классификации.

Разберём, как применять предобученную на *ImageNet* сеть. Чтобы адаптировать *ResNet50* к нашей задаче, уберём верхушку и сконструируем

её заново:

```
from tensorflow.keras.layers import GlobalAveragePooling2D, Dense
from tensorflow.keras.models import Sequential

backbone = ResNet50(input_shape=(150, 150, 3),
                    weights='imagenet',
                    include_top=False)

model = Sequential()
model.add(backbone)
model.add(GlobalAveragePooling2D())
model.add(Dense(12, activation='softmax'))
```

где `backbone` (англ. «костяк») — то, что осталось от *ResNet50*.

Научим вас трюку. Допустим, есть очень маленький датасет: всего 100 картинок и два класса. Если на таком датасете обучить *ResNet50*, то она гарантированно переобучится: в ней слишком много параметров — порядка 23 млн! У сети будут идеальные предсказания на обучающей выборке и случайные — на тестовой.

Чтобы этого избежать, «заморозим» часть сети: некоторые слои оставим с весами из *ImageNet*, они не будут обучаться градиентным спуском. Обучим только 1–2 полносвязных слоя наверху сети. Так количество параметров в сети уменьшится, но архитектура сохранится.

Заморозим сеть так:

```
backbone = ResNet50(input_shape=(150, 150, 3),
                    weights='imagenet',
                    include_top=False)

# замораживаем ResNet50 без верхушки
backbone.trainable = False

model = Sequential()
model.add(backbone)
model.add(GlobalAveragePooling2D())
model.add(Dense(12, activation='softmax'))
```

Чтобы сеть обучалась, добавленный сверху *backbone* полносвязный слой замораживать не стали.

Заморозка позволяет избавиться от переобучения и повысить скорость обучения сети: градиентному спуску считать производные для замороженных слоёв не нужно.



