# Сверточные сети и Transfer learning

Цель этого ноутбука - знакомство со сверточными сетями и transfer learning на примере классификации картинок.
План семинара.

- Конволюция, Pooling — базовые слои, их гиперпараметры и интуиция использования.
- Построение сверточной нейросети для классификации картинок.
- Применение аугментации для улучшения качетсва.
- Использование Transfer Learning для этой же задачи.
- Получение нейросетевых дескрипторов.

In [1]:

```
import os
import time
import glob
import requests
from tqdm.notebook import tqdm
from collections import defaultdict

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split

import torch
from torch import nn
import torch.nn.functional as F

import torchvision
from torchvision import transforms

from IPython.display import clear_output
%matplotlib inline

sns.set(font_scale=1.7, style='darkgrid', palette='Set2')

# device_num = 0
# torch.cuda.set_device(device_num)
```

In [2]:

```
device = f"cuda" if torch.cuda.is_available() else "cpu"
# device = "cpu"
print(device)
```

```
cuda
```

## 1. Convolution (свёртка)

Основные гиперпараметры:

- `in_channels` (int) - количество каналов во входном изображении
- `out_channels` (int) - количество каналов после применения свертки (кол-во ядер (фильтров), которые будут применены)
- `kernel_size` (int, tuple) - размер сверточного ядра
- `stride` (int, tuple) - шаг, с которым будет применена свертка. Значение по умолчанию 1
- `padding` (int, tuple) - добавление по краям изображения дополнительных пикселей. Значение по умолчанию 0
- `padding_mode` (string, optional) - принцип заполнения краёв. Значение по умолчанию 'zeros'

`nn.Conv2d(in_channels=1, out_channels=1, kernel_size=3, stride=2, padding=1, padding_mode='zeros')`

Берем `out_channels` фильтров размера `in_channels x kernel_size x kernel_size`. Каждым фильтром 'проходим' по изображению с шагом `stride`, поэлементно умножаем его на область изображения размером `in_channels x kernel_size x kernel_size`, складываем получившиеся поэлемнетные произведения и записываем это число в результирующий тензор. В итоге получаем `out_channels` выходных тензоров.

**Интуиция**:

В FC слоях мы соединяли нейрон с каждым нейроном на предыдущем слое. Теперь нейрон соединен только с ограниченной областью выхода предыдущего слоя. Иногда эту область называют *рецептивном полем (receiptive field)* нейрона.

Такое изменение необходимо из-за большой размерности входных данных. Например, если размер входного изображения $3 \cdot 224 \cdot 224$, то каждый нейроне в FC-слое будет содержать $3 \cdot 224 \cdot 224 = 150\,528$ параметров, что очень много. При этом мы захотим добавить нелинейности в нашу архитектуру, так что у нас будет несколько таких слоёв.

**Вопрос**

К изображению (3, 224, 224) применяют свертку `nn.Conv2d(in_channels=3, out_channels=64, kernel_size=5, stride=2, padding=2)`.

- Какой будет размер выходного изображения?
- Сколько у данного слоя обучаемых параметров?

*Проверяем себя.* Для этого будем использовать библиотеку torchinfo (в прошлом torchsummary). Метод summary данной библиотеки позволяет визуализировать основные характеристики нейронной сети.

In [3]:

```
! pip3 install torchinfo
from torchinfo import summary
```

```
Collecting torchinfo
  Downloading torchinfo-1.7.1-py3-none-any.whl (22 kB)
Installing collected packages: torchinfo
Successfully installed torchinfo-1.7.1
WARNING: Running pip as the 'root' user can result in broken permissio
ns and conflicting behaviour with the system package manager. It is re
commended to use a virtual environment instead: https://pip.pypa.io/wa
rnings/venv (https://pip.pypa.io/warnings/venv)
```

In [4]:

```
model = nn.Sequential()
model.add_module('conv', nn.Conv2d(in_channels=3, out_channels=64,
                                   kernel_size=5, stride=2, padding=2))

summary(model.to(device), (2, 3, 224, 224))
```

Out[4]:

```
==========================================================================
====================
Layer (type:depth-idx)                   Output Shape              Par
am #
==========================================================================
====================
Sequential                               [2, 64, 112, 112]         --
├─Conv2d: 1-1                            [2, 64, 112, 112]         4,8
64
==========================================================================
====================
Total params: 4,864
Trainable params: 4,864
Non-trainable params: 0
Total mult-adds (M): 122.03
==========================================================================
====================
Input size (MB): 1.20
Forward/backward pass size (MB): 12.85
Params size (MB): 0.02
Estimated Total Size (MB): 14.07
==========================================================================
====================
```

In [5]:

```
3 * 64 * 25 + 64
```

Out[5]:

4864

Посмотрим на то, как применение свёртки с определёнными фильтрами влияет на изображение и как будет меняться картинка в зависимости от фильтра:

In [6]:

```
! wget https://www.kotzendes-einhorn.de/blog/wp-content/uploads/2011/01/lenna.j
```

```
--2022-11-29 01:58:52--  https://www.kotzendes-einhorn.de/blog/wp-cont
ent/uploads/2011/01/lenna.jpg (https://www.kotzendes-einhorn.de/blog/w
p-content/uploads/2011/01/lenna.jpg)
Resolving www.kotzendes-einhorn.de (www.kotzendes-einhorn.de)... 94.13
0.145.107
Connecting to www.kotzendes-einhorn.de (www.kotzendes-einhorn.de)|94.1
30.145.107|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 64098 (63K) [image/jpeg]
Saving to: ‘lenna.jpg’

lenna.jpg           100%[===================>]  62.60K   221KB/s    in
0.3s

2022-11-29 01:58:54 (221 KB/s) - ‘lenna.jpg’ saved [64098/64098]
```

In [7]:

```
img = plt.imread('./lenna.jpg')

plt.figure(figsize=(12,7))
plt.imshow(img)
plt.axis("off");
```

Функция для инициализации весов слоя.

In [8]:

```
def init_conv(kernel):
    conv = nn.Conv2d(
        in_channels=3, out_channels=1,
        kernel_size=3, bias=False
    )
    conv.weight = torch.nn.Parameter(
        torch.FloatTensor(kernel),
        requires_grad=False
    )
    return conv
```

Функция для свертки изображения с одним фильтром.

In [9]:

```
def convolution(kernel, img, transforms):
    conv = init_conv(kernel)
    img_tensor = transform(img)
    # Добавим батч-размерность
    res = conv(img_tensor.unsqueeze(0))
    # Избавимся не единичных разменостей
    res = res.detach().squeeze()
    # Пиксели имеют значения от 0 до 255
    res = (torch.clip(res, 0, 1) * 255).int()
    return res
```

Numpy-картинку нужно привести к torch-тензору. Кроме того, для визуализации нам нужно отобразить картинку в черно-белом цвете. Это можно сделать с помощью модуля transforms. Подробнее будет в разделе про аугментации.

In [10]:

```
transform = transforms.Compose([
    transforms.ToTensor(),   # Переводим массив в торч-тензор
    transforms.Grayscale(),  # Делаем изображение черно-белым
])
```

Зададим 2 фильтра.

In [11]:

```
kernel_1 = torch.FloatTensor([[[
    [-1, 0, 1],
    [-2, 0, 2],
    [-1, 0, 1]
]]])

kernel_2 = torch.FloatTensor([[[
    [-1, -2, -1],
    [0, 0, 0],
    [1, 2, 1]
]]])
```

In [12]:

```
fig, axs = plt.subplots(
    nrows=1, ncols=2, figsize=(16, 7),
    sharey=True, sharex=True
)

res_images = []
for ax, kernel in zip(axs.flatten(), [kernel_1, kernel_2]):
    res = convolution(kernel, img, transform)
    ax.imshow(res, cmap='gray')
    ax.grid(b=None)
    ax.set_xticks([])
    ax.set_yticks([])
    res_images.append(res)
plt.suptitle('Фильтры (операторы) Собеля')
plt.show()
```

```
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:10: Matpl
otlibDeprecationWarning: The 'b' parameter of grid() has been renamed
'visible' since Matplotlib 3.5; support for the old name will be dropp
ed two minor releases later.
  # Remove the CWD from sys.path while we load stuff.
```


Чем более пиксель белый, тем больше его значение. Если присмотреться, то можно заметить, как на первом результате фильтр (ядро) делает более значимыми (белыми) пиксели, соответствующие вертикальным линиям: нос, полоска справа, волосы, а на втором — горизонтальным: брови, губы.

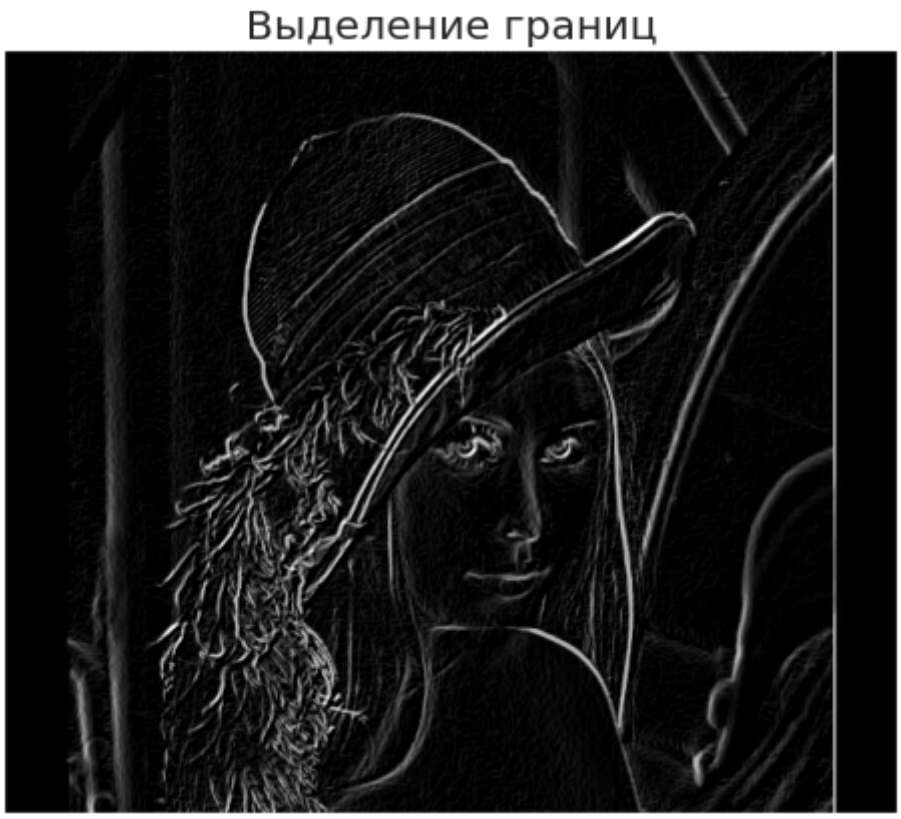
Это как раз согласуется со значениями в фильтрах (ядрах): первый вычисляет перепады значений в пикселях по вертикали, второй — по горизонтали.

С помощью этих фильтров легко прийти к методу выделения границ на изображении: поскольку каждая граница состоит из $x$ и $y$ компоненты, то используем теорему Пифагора и вычислим суммарное значение:

In [13]:

```
img_sobel = np.sqrt(res_images[0] ** 2 + res_images[1] ** 2)

plt.figure(figsize=(12,7))
plt.title('Выделение границ')
plt.imshow(img_sobel, cmap='gray')
plt.grid(b=None)
plt.xticks([])
plt.yticks([])
plt.show()
```

```
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:6: Matplo
tlibDeprecationWarning: The 'b' parameter of grid() has been renamed
'visible' since Matplotlib 3.5; support for the old name will be dropp
ed two minor releases later.
```




Полученный фильтр также [оператором Собеля](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_%D0%)
Так, мы посмотрели как работает свертка на примере оператора Собеля.

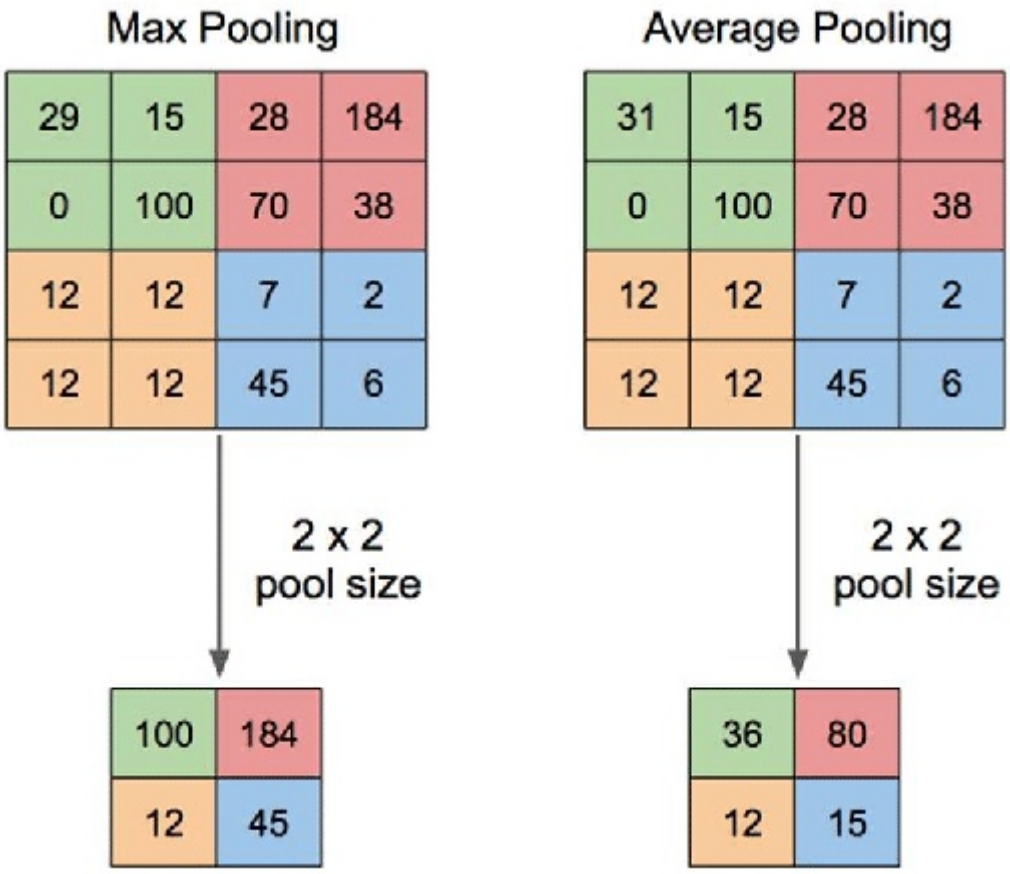
## 2. Pooling

Основные гиперпараметры:

- `kernel_size` (int, tuple) - размер ядра
- `stride` (int, tuple) - шаг, с которым будет применен pooling. Значение по умолчанию `kernel_size`
- `padding` (int, tuple) - добавление по краям изображения нулей

Основные виды pooling-ов:

- `MaxPooling` — берется максимум элементов,
- `AveragePooling` — берется среднее элементов.




**Интуиция:**

- снижаем размерность изображения и, как следствие, вычислительную сложность;
- увеличиваем рецептивное поле на входном изображении для нейронов следующих сверточных слоев.

При этом многие исследователи ставят под сомнение эффективность pooling слоёв. Например, в статье [Striving for Simplicity: The All Convolutional Net (https://arxiv.org/abs/1412.6806)](https://arxiv.org/abs/1412.6806) предлагается заменить его на свертки с большим stride-ом. Также считается, что отсутствие pooling слоёв хорошо сказывается на обучении генеративных моделей, но споры ещё ведутся: [FCC-GAN: A Fully Connected and Convolutional Net Architecture for GANs (https://arxiv.org/pdf/1905.02417.pdf)](https://arxiv.org/pdf/1905.02417.pdf)

**Вопрос:** сколько параметров у pooling слоя?

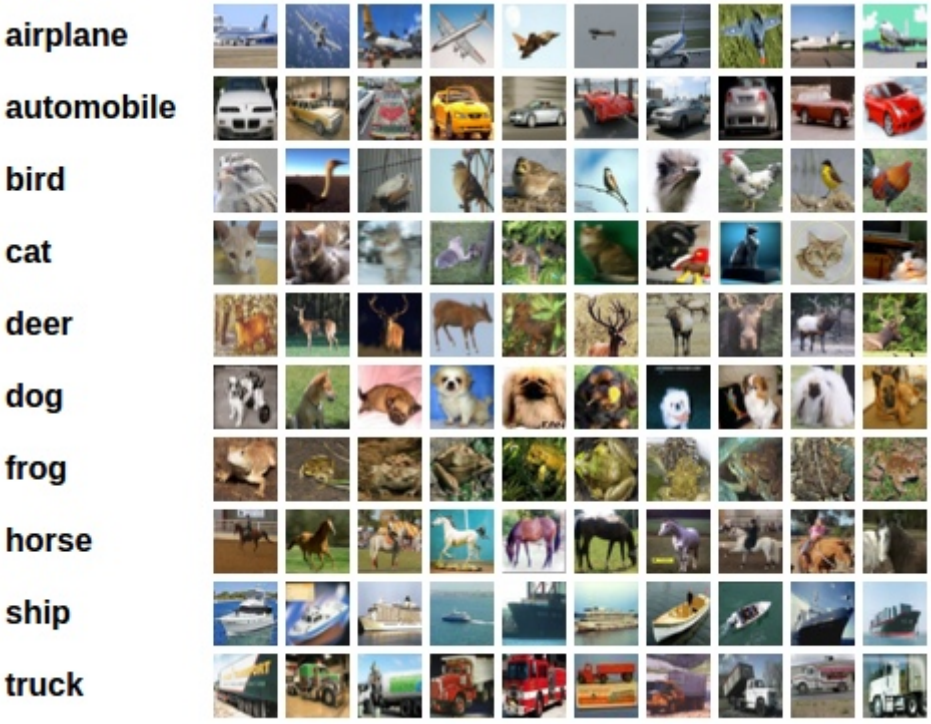
Применим наши знания для решения конкретной задачи.

## 3. CIFAR10

Датасет состоит из 60k картинок 32x32x3.

50k — обучающая выборка, 10k — тестовая.

10 классов: 'plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'.




Загружаем датасет.

In [14]:

```
# Часть данных для обучения
train_dataset = torchvision.datasets.CIFAR10(
    root='./cifar', download=True, train=True, transform=transforms.ToTensor())

# Валидационная / тестовая часть данных
val_dataset = torchvision.datasets.CIFAR10(
    root='./cifar', download=True, train=False, transform=transforms.ToTensor()

# Классы объектов в датасете
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz (h
ttps://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz) to ./cifar/cif
ar-10-python.tar.gz

  0%|          | 0/170498071 [00:00<?, ?it/s]

Extracting ./cifar/cifar-10-python.tar.gz to ./cifar
Files already downloaded and verified
```

Инициализируем генераторы батчей:

In [15]:

```
batch_size = 64

train_batch_gen = torch.utils.data.DataLoader(train_dataset, batch_size=batch_s
val_batch_gen = torch.utils.data.DataLoader(val_dataset, batch_size=batch_size,
```

Пайплайн обучения.

In [16]:

```python
def plot_learning_curves(history):
    '''
    Функция для вывода лосса и метрики во время обучения.

    :param history: (dict)
        accuracy и loss на обучении и валидации
    '''

    plt.figure(figsize=(20, 7))

    plt.subplot(1,2,1)
    plt.title('Лосс')
    plt.plot(history['loss']['train'], label='train', lw=3)
    plt.plot(history['loss']['val'], label='val', lw=3)
    plt.xlabel('Эпоха')
    plt.legend()

    plt.subplot(1,2,2)
    plt.title('Точность')
    plt.plot(history['acc']['train'], label='train', lw=3)
    plt.plot(history['acc']['val'], label='val', lw=3)
    plt.xlabel('Эпоха')
    plt.legend()

    plt.show()
```

In [17]:

```python
def train(
    model,
    criterion,
    optimizer,
    train_batch_gen,
    val_batch_gen,
    num_epochs=50
):
    '''
    Функция для обучения модели и вывода лосса и метрики во время обучения.

    :param model: обучаемая модель
    :param criterion: функция потерь
    :param optimizer: метод оптимизации
    :param train_batch_gen: генератор батчей для обучения
    :param val_batch_gen: генератор батчей для валидации
    :param num_epochs: количество эпох

    :return: обученная модель
    :return: (dict) accuracy и loss на обучении и валидации ("история" обучения
    '''

    history = defaultdict(lambda: defaultdict(list))

    for epoch in range(num_epochs):
        train_loss = 0
        train_acc = 0
        val_loss = 0
        val_acc = 0

        start_time = time.time()

        # Устанавливаем поведение dropout / batch_norm  в обучение
        model.train(True)

        # На каждой "эпохе" делаем полный проход по данным
        for X_batch, y_batch in train_batch_gen:
            # Обучаемся на батче (одна "итерация" обучения нейросети)
            X_batch = X_batch.to(device)
            y_batch = y_batch.to(device)

            # Логиты на выходе модели
            logits = model(X_batch)

            # Подсчитываем лосс
            loss = criterion(logits, y_batch.long().to(device))

            # Обратный проход
            loss.backward()
            # Шаг градиента
            optimizer.step()
            # Зануляем градиенты
            optimizer.zero_grad()

            # Сохраняем лоссы и точность на трейне
            train_loss += loss.detach().cpu().numpy()
            y_pred = logits.max(1)[1].detach().cpu().numpy()
            train_acc += np.mean(y_batch.cpu().numpy() == y_pred)

```

```
        # Подсчитываем лоссы и сохраняем в "историю"
        train_loss /= len(train_batch_gen)
        train_acc /= len(train_batch_gen)
        history['loss']['train'].append(train_loss)
        history['acc']['train'].append(train_acc)

        # Устанавливаем поведение dropout / batch_norm в режим тестирования
        model.train(False)

        # Полный проход по валидации
        for X_batch, y_batch in val_batch_gen:
            X_batch = X_batch.to(device)
            y_batch = y_batch.to(device)

            # Логиты, полученные моделью
            logits = model(X_batch)

            # Лосс на валидации
            loss = criterion(logits, y_batch.long().to(device))

            # Сохраняем лоссы и точность на валидации
            val_loss += loss.detach().cpu().numpy()
            y_pred = logits.max(1)[1].detach().cpu().numpy()
            val_acc += np.mean(y_batch.cpu().numpy() == y_pred)

        # Подсчитываем лоссы и сохраняем в "историю"
        val_loss /= len(val_batch_gen)
        val_acc /= len(val_batch_gen)
        history['loss']['val'].append(val_loss)
        history['acc']['val'].append(val_acc)

        clear_output()

        # Печатаем результаты после каждой эпохи
        print("Epoch {} of {} took {:.3f}s".format(
            epoch + 1, num_epochs, time.time() - start_time))
        print("  training loss (in-iteration): \t{:.6f}".format(train_loss))
        print("  validation loss (in-iteration): \t{:.6f}".format(val_loss))
        print("  training accuracy: \t\t\t{:.2f} %".format(train_acc * 100))
        print("  validation accuracy: \t\t\t{:.2f} %".format(val_acc * 100))

        plot_learning_curves(history)

    return model, history
```

### 3.1. Baseline

Начнем с простой линейной модели, рассмотренной на прошлом семинаре по нейросетям:

In [18]:

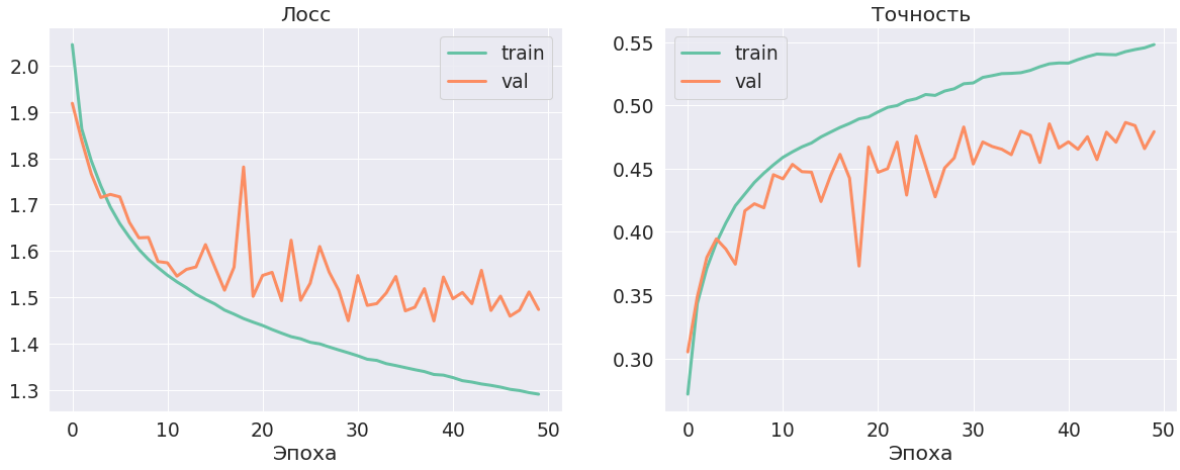
```python
class MySimpleModel(nn.Module):
    def __init__(self):
        '''
        Здесь объявляем все слои, которые будем использовать
        '''

        super(MySimpleModel, self).__init__()
        # входное количество признаков = высота * ширина * кол-во каналов карти
        # сейчас 64 нейрона в первом слое
        self.linear1 = nn.Linear(3 * 32 * 32, 64)
        # 10 нейронов во втором слое
        self.linear2 = nn.Linear(64, 10)  # логиты (logits) для 10 классов

    def forward(self, x):
        '''
        Здесь пишем в коде, в каком порядке какой слой будет применяться
        '''

        x = self.linear1(nn.Flatten()(x))
        x = self.linear2(nn.ReLU()(x))
        return x
```

Применим ее к нашим данным — картинками из CIFAR10:

In [19]:

```python
model = MySimpleModel().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

model, history = train(
    model, criterion, optimizer,
    train_batch_gen, val_batch_gen,
    num_epochs=50
)
```

```
Epoch 50 of 50 took 8.317s
  training loss (in-iteration):         1.290378
  validation loss (in-iteration):       1.473276
  training accuracy:                    54.82 %
  validation accuracy:                  47.93 %
```



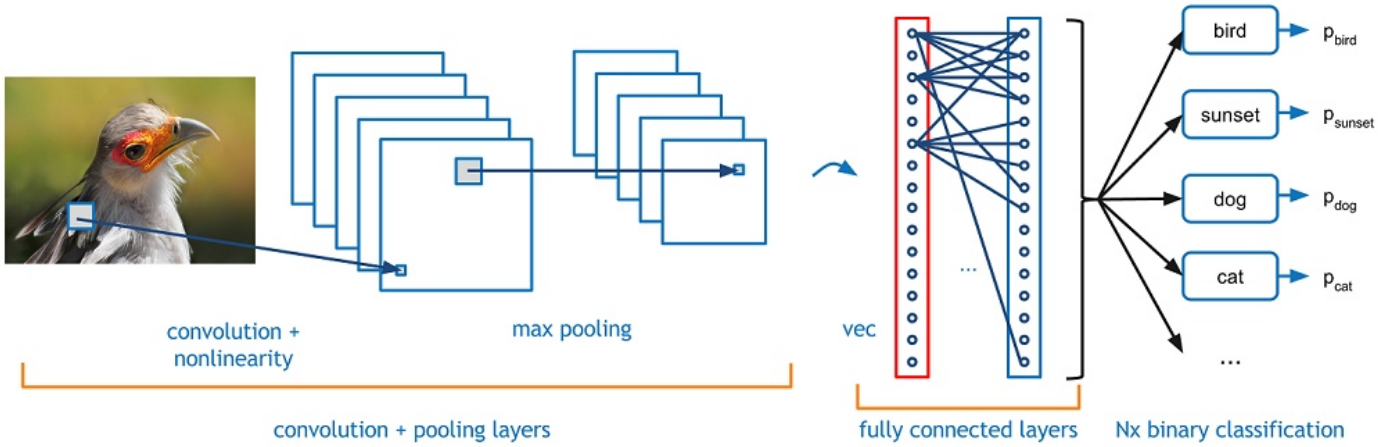
## 3.2. Свёрточная нейросеть

*Свёрточная нейросеть / Convolutional Neural Network / CNN* — это многослойная нейросеть, имеющая в своей архитектуре свёрточные слои / Conv Layers и pooling-слои / Pool Layers.

Простые свёрточные нейросети для классификации, почти всегда строятся по следующему правилу:

$$INPUT \rightarrow [[CONV->RELU]^N \rightarrow POOL?]^M \rightarrow [FC->RELU]^K \rightarrow FC$$

"?" обозначает опциональные слои.




Подробнее:

1. Входной слой (batch картинок HxWxC)
2. $M$ блоков ($M \geq 0$) из свёрток и pooling-ов. Все эти $M$ блоков вместе называют *feature extractor* свёрточной нейросети, потому что эта часть сети отвечает непосредственно за формирование новых, более сложных признаков, поверх тех, которые подаются.

   При этом лучше использовать несколько сверток с маленьким рецептивным полем, чем одну свертку с большим рецептивным полем.
3. $K$ штук FullyConnected-слоёв с активациями. Эту часть из $K$ FC-слоёв называют *classificator*, поскольку эти слои отвечают непосредственно за предсказание нужно класса.

   *Замечание*: Pooling layer можно пропустить и не включать в архитектуру, но при этом он снижает размерность, а следовательно и вычислительную сложность, а также помогает бороться с переобучение.

Также нужно не забывать о пользе `Dropout` и `BatchNorm`:

- `Dropout` позволяет бороться с переобучением, можно интерпретировать как обучение ансамбля моделей.
- `BatchNorm` нормирует данные, делает веса на более поздних слоях менее чувствительными к изменениям весов на начальных слоях. Таким образом `BatchNorm` позволяет сделать нейросеть более стабильной при изменении распределения входных данных.

**Вопрос**: посмотрите на следующую нейросеть и укажите на некорректные шаги в реализации.

In [20]:

```
model = nn.Sequential()
model.add_module('conv1', nn.Conv2d(3, 2048, kernel_size=5, stride=2, padding=3
model.add_module('mp1', nn.MaxPool2d(7))
model.add_module('conv2', nn.Conv2d(2048, 64, kernel_size=3))
model.add_module('mp2', nn.MaxPool2d(2))
model.add_module('bn1', nn.BatchNorm2d(64))
model.add_module('dp1', nn.Dropout(0.5))
model.add_module('relu1', nn.ReLU())

model.add_module('conv3', nn.Conv2d(64, 128, kernel_size=(20, 20)))
model.add_module('mp3', nn.MaxPool2d(2))
model.add_module('conv4', nn.Conv2d(128, 256, kernel_size=(20, 20)))

model.add_module('flatten', nn.Flatten())
model.add_module('fc1', nn.Linear(1024, 512))
model.add_module('fc2', nn.Linear(512, 10))
model.add_module('dp2', nn.Dropout(0.05))
```

> Подсказка:

*(нужно дважды кликнуть на ячейку)*

Исправим все ошибки и обучим полученную сверточную сеть.

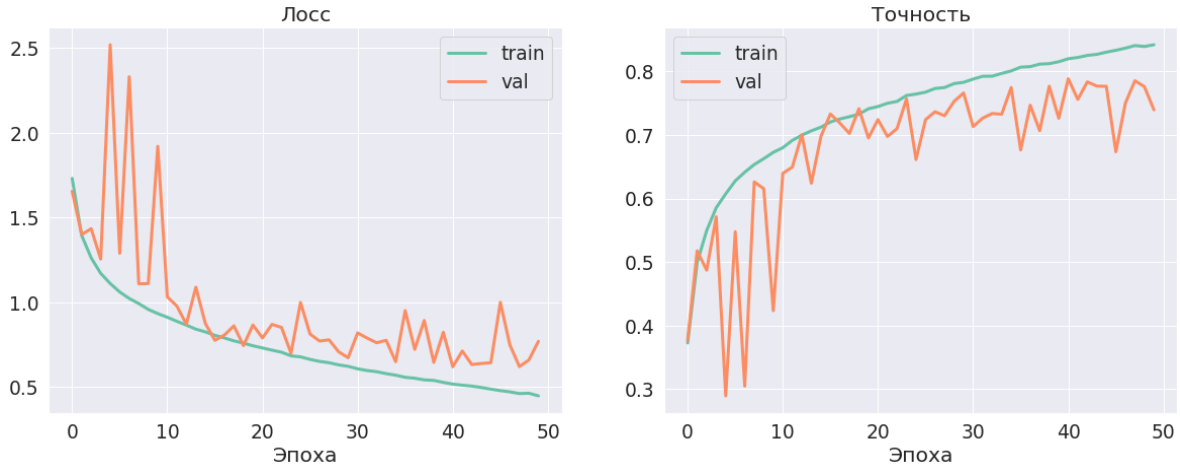
In [21]:

```
class SimpleConvNet(nn.Module):
    def __init__(self):
        super(SimpleConvNet, self).__init__()

        self.conv1 = nn.Conv2d(3, 32, 3)
        self.mp1 = nn.MaxPool2d(2)
        self.bn1 = nn.BatchNorm2d(32)
        self.droupout1 = nn.Dropout(0.3)
        self.relu1 = nn.ReLU()

        self.conv2 = nn.Conv2d(32, 64, 3)
        self.mp2 = nn.MaxPool2d(2)
        self.bn2 = nn.BatchNorm2d(64)
        self.droupout2 = nn.Dropout(0.3)
        self.relu2 = nn.ReLU()

        self.flatten = nn.Flatten()
        self.fc3 = nn.Linear(2304, 512)
        self.droupout3 = nn.Dropout(0.3)
        self.relu3 = nn.ReLU()
        self.fc4 = nn.Linear(512, 10)

    def forward(self, x):
        layer1 = self.mp1(self.conv1(x))
        layer1 = self.relu1(self.droupout1(self.bn1(layer1)))

        layer2 = self.mp2(self.conv2(layer1))
        layer2 = self.relu2(self.droupout2(self.bn2(layer2)))

        out = self.flatten(layer2)
        out = self.relu3(self.droupout3(self.fc3(out)))
        out = self.fc4(out)
        return out
```

In [22]:

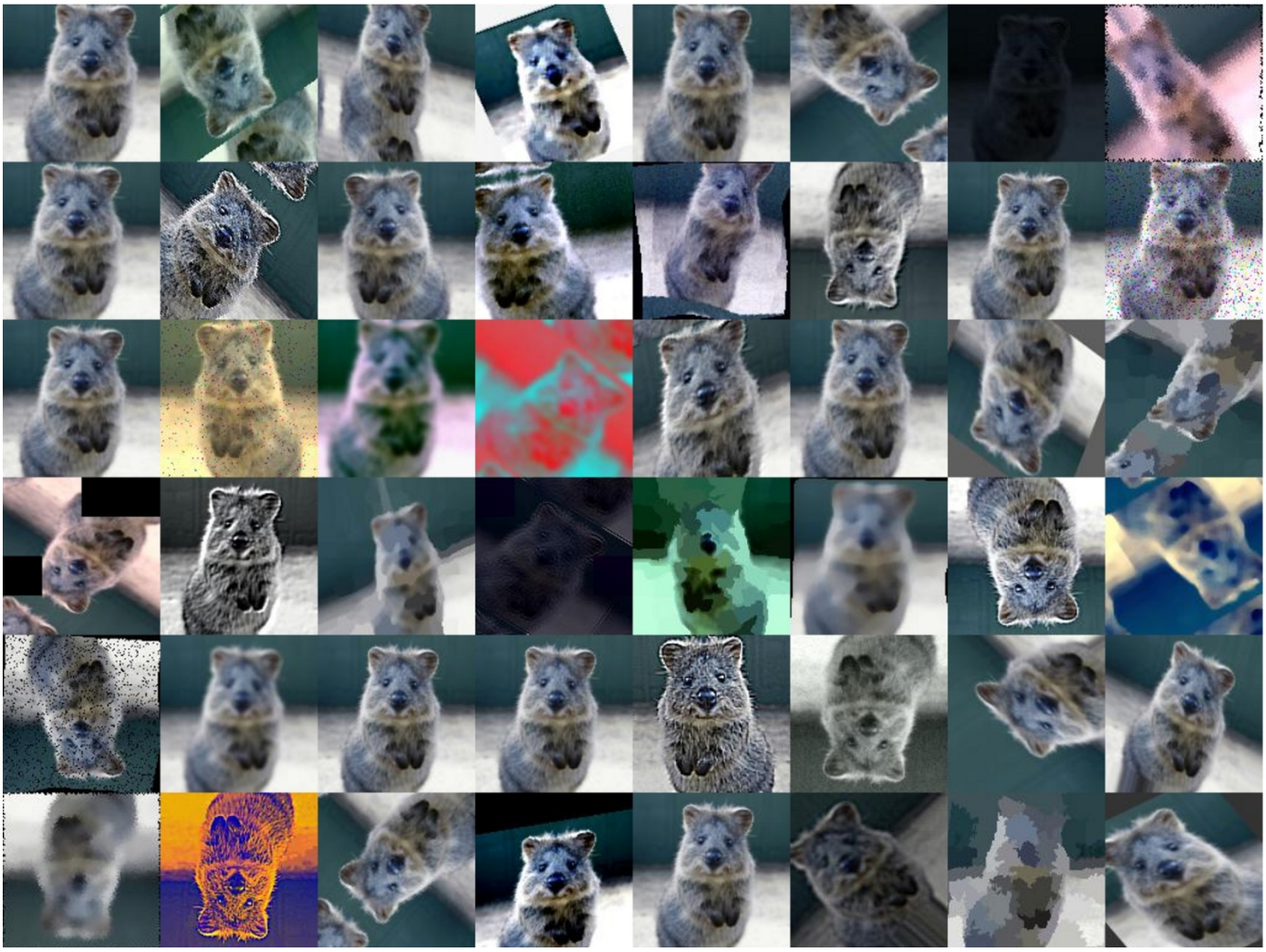
```
model = SimpleConvNet().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

model, history = train(
    model, criterion, optimizer,
    train_batch_gen, val_batch_gen,
    num_epochs=50
)
```

```
Epoch 50 of 50 took 9.708s
  training loss (in-iteration):         0.446676
  validation loss (in-iteration):       0.768785
  training accuracy:                    84.21 %
  validation accuracy:                  73.96 %
```




Сравните полученное качество с тем, что мы получили ранее для полносвязной сети.

## 4. Аугментация



**Дополнение данных / Аугментация данных / Data augmentations** — это метод, направленный на увеличение размеров обучающей выборки. Дополнение обучающей выборки разнообразными, "хорошими" и "плохими" примерами, позволяет получить модель более устойчивую на тестовых данных, так как для неё в тестовых данных будет меньше "неожиданностей".

В данном ноутбуке для аугментаций будет использоваться модуль `torchvision.transforms` . Однако это не единственный способ для работы с аугментациями. Существуют такие библиотеки как Scikit-Image (https://scikit-image.org/), Augmentor (https://augmentor.readthedocs.io/en/master/), imgaug (https://imgaug.readthedocs.io/en/latest/), Albumentations (https://albumentations.ai/) и др, которые также могут быть использованы для аугментации изображений.

Благодаря модулю `torchvision.transforms` агументации можно делать очень просто. Про все реализованные в библиотеке преобразования можно почитать здесь (https://pytorch.org/docs/stable/torchvision/transforms.html). Мы рассмотрим наиболее распространенные классы аугментаций.

- `RandomAffine(degrees, translate=None, scale=None, shear=None, interpolation=<InterpolationMode.NEAREST: 'nearest'>, fill=0, fillcolor=None, resample=None)` — **случайное афинное преобразование** с сохранением центра.
  - `degrees` — градус вращения.
  - `translate` — смещение.
  - `scale` — масштаб.

- `ColorJitter(brightness=0, contrast=0, saturation=0, hue=0)` — *случайное* изменение **яркости** / brightness, **контраста** / contrast, **насыщенности** / saturation и **тонов** / hue цветов. Если на вход приходит torch.Tensor, то его размерность дожна быть `[..., 3, H, W]` . Если PIL.Image, то

без альфа-канала. Каждый из параметров может быть быть задан в виде float числа: param, или пары float чисел: min, max. Значение парметра выбирается случайно из отрезка `[1 - param, 1 + param]` или `[min, max]` для brightness, contrast, saturation. Значение парметра должно быть нотрицательным. Значение парметра hue выбирается случайно из отрезка `[-hue, hue]` или `[min, max]`. При этом значение 0<= `hue` <= `0.5` or `-0.5` <= `min` <= `max` <= `0.5`.

- `CenterCrop(size)` — вырезает **прямоугольную область** размером `size[0] x size[1]`, если size задан туплом, если `size` задан числом — `size x size` **из центра картинки**.
- `GaussianBlur(kernel_size, sigma)` — *случайное* **гауссовское размытие изображения**. `kernel_size` — размер гауссовского ядра. `sigma` — стандартное отклонение. `sigma` может быть задано в виде чила, тогда парметр фиксирован, или в виде тупла in, max, тогда оно выбирается случайно из отрезка `[min, max]`.
- `Grayscale(num_output_channels=1)` и `RandomGrayscale` (p=0.1) — неслучайная и *случайная* **трансформации картинки в ч/б формат**. Grayscale имеет парметр `num_output_channels`, который означет количество каналов на выходе, он может быть равен 1 или 3. RandomGrayscale имеет парметр p, который равен вероятности применения преобразования. Тензор на выхоже будет иметь столько же каналов, сколько тензор на входе.
- `Normalize(mean, std, inplace=False)` — **нормализация тензора картинки** с заданными средним и отклонением для каждого канала. То есть `mean = (mean[1], ..., mean[n])`, `std = (std[1], ..., std[n])`, где `n` — количество каналов. Не поддерживает PIL.Image формат!
- `RandomResizedCrop(size, scale=(0.08, 1.0), ratio=(0.75, 1.3333333333333333), interpolation=<InterpolationMode.BILINEAR: 'bilinear'>)` — **случайное обрезание картинки** со случайным выбором размера и соотношения сторон и последующим **увеличеним картинки до первонального размера**.
- `Resize(size, interpolation=<InterpolationMode.BILINEAR: 'bilinear'>)` — **изменение размеров кратинки**. Если `size` задан числом, то наименьшая из размерностей картинки приобретает размер `size`. Иначе, если размер задан парой, то размер картинки становится равным `size[0] x size[1]`.

Для того, чтобы получить преобразование, которого нет в модуле torchvision.transforms можно использовать `Lambda` преобразование. Например, получить гауссовский шум на изображении можно так:

```
Lambda(lambda x : x + torch.randn_like(x))
```

Выше перечисленные трансформации применяются к данным типа PIL.Image или torch.Tensor,на выходе выдают соотвествующий формат. Для того, чтобы в через тарнсформации получить PIL.Image, можно использовать класс `ToPILImage`, для того, чтобы получить torch.Tensor — `ToTensor`. Эти классы в методе forward могут использовать torch.Tensor, np.ndarray и PIL.Image, np.ndarray соотвественно.

Чтобы объединить несколько трансформаций можно использовать `Compose` (transforms), где transforms — список из объектов коассов преобразований.

Применим несколько случайных преобразований к картинкам, и тем самым расширим нашу выборку. Так как `CIFAR` выдает `PIL.Image`, то необходимо применить преобразование `ToTensor`.

In [23]:

```
from torchvision import transforms

# Набор аугментаций при обучении
transform_train = transforms.Compose([
    transforms.ColorJitter(brightness=0.9, contrast=0.9, saturation=0.9),
    transforms.RandomAffine(degrees=30),
    transforms.RandomResizedCrop(size=(32, 32)),
    transforms.ToTensor(),
])

# Набор аугментаций при валидации и тестировании
transform_val = transforms.Compose([
    transforms.ToTensor(),
])
```

Загружаем данные, устанавливаем ранее заданные трансформации для обучения и валидации.

In [24]:

```
# Часть данных для обучения
train_dataset = torchvision.datasets.CIFAR10(
    root='./cifar', download=True, train=True, transform=transform_train)

# Валидационная / тестовая часть данных
val_dataset = torchvision.datasets.CIFAR10(
    root='./cifar', download=True, train=False, transform=transform_val)

```

```
Files already downloaded and verified
Files already downloaded and verified
```

Перезагружаем даталоадеры.

In [25]:

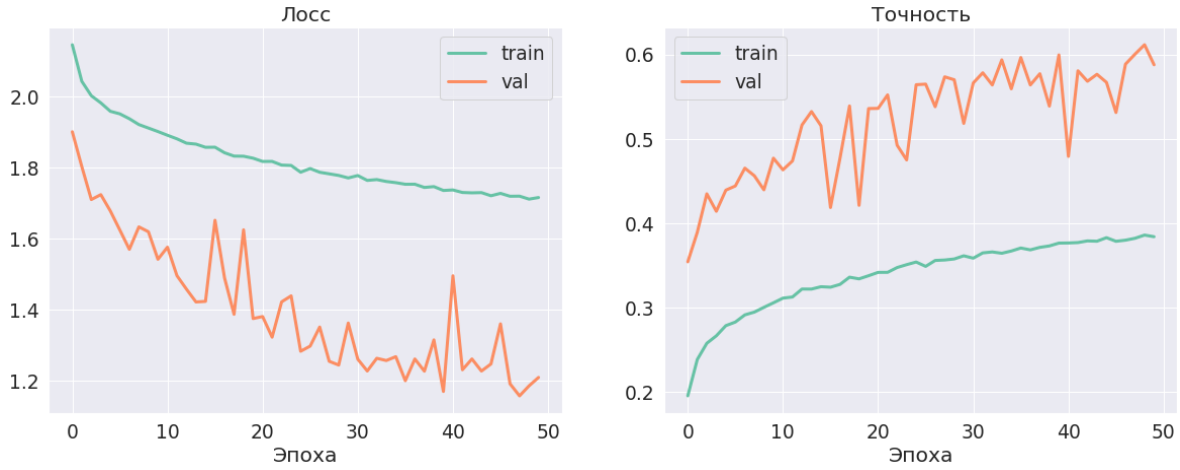
```
batch_size = 64

train_batch_gen = torch.utils.data.DataLoader(train_dataset, batch_size=batch_s
val_batch_gen = torch.utils.data.DataLoader(val_dataset, batch_size=batch_size,
```

Обучим нейросеть на новых данных.

In [26]:

```
model = SimpleConvNet().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

model, history = train(
    model, criterion, optimizer,
    train_batch_gen, val_batch_gen,
    num_epochs=50
)
```

```
Epoch 50 of 50 took 39.980s
  training loss (in-iteration):         1.715039
  validation loss (in-iteration):       1.209284
  training accuracy:                    38.38 %
  validation accuracy:                  58.76 %
```




Каждую эпоху для каждого изображения выбирается случайная трансформация. Таким образом каждую эпоху нейросеть обучается на одном и том же количестве изображений, но они каждый раз разные.

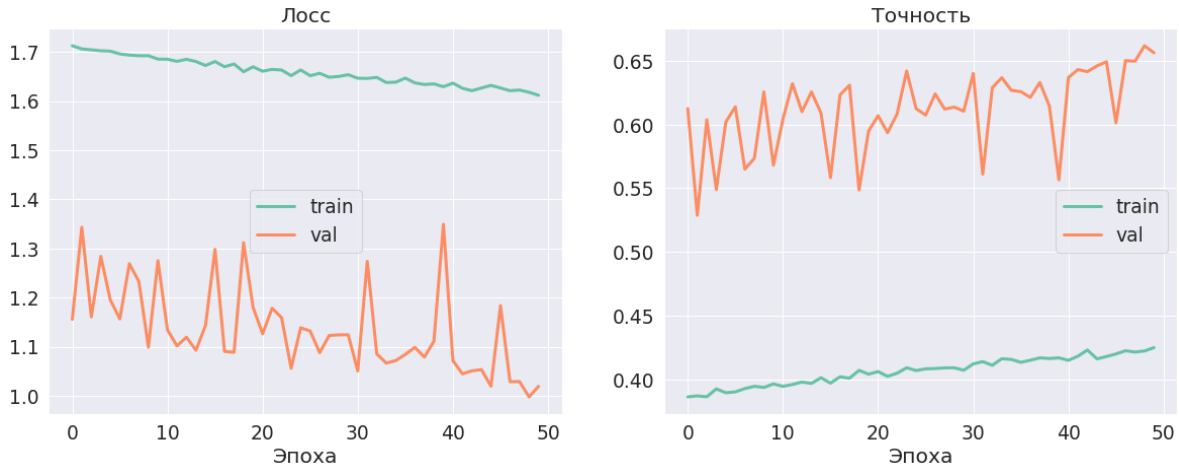
> Сравните, как аугментация влияет на качество на обучении и на валидации.

Запустим еще 50 эпох.

In [27]:

```
model, history = train(
    model, criterion, optimizer,
    train_batch_gen, val_batch_gen,
    num_epochs=50
)
```

```
Epoch 50 of 50 took 60.268s
  training loss (in-iteration):         1.611806
  validation loss (in-iteration):       1.019204
  training accuracy:                    42.49 %
  validation accuracy:                  65.63 %
```




Аугментации уменьшают скорость обучения, но могут хорошо расширить исходную выборку данных.

# 5. Transfer Learning

Transfer Learning (https://arxiv.org/abs/1808.01974v) — это процесс дообучения на *новых данных* какой-либо нейросети, предобученной до этого на других данных. Обычно предобучение производят на хорошем, большом датасете размером около миллиона картинок, например, ImageNet ~ 14 млн картинок.

На данный момент есть множество предобученных моделей: `AlexNet`, `DenseNet`, `ResNet`, `VGG`, `Inception` и другие, а также их различные модификации. Все они отличаются архитектурой и входными данными.

## 5.1 Описание метода:

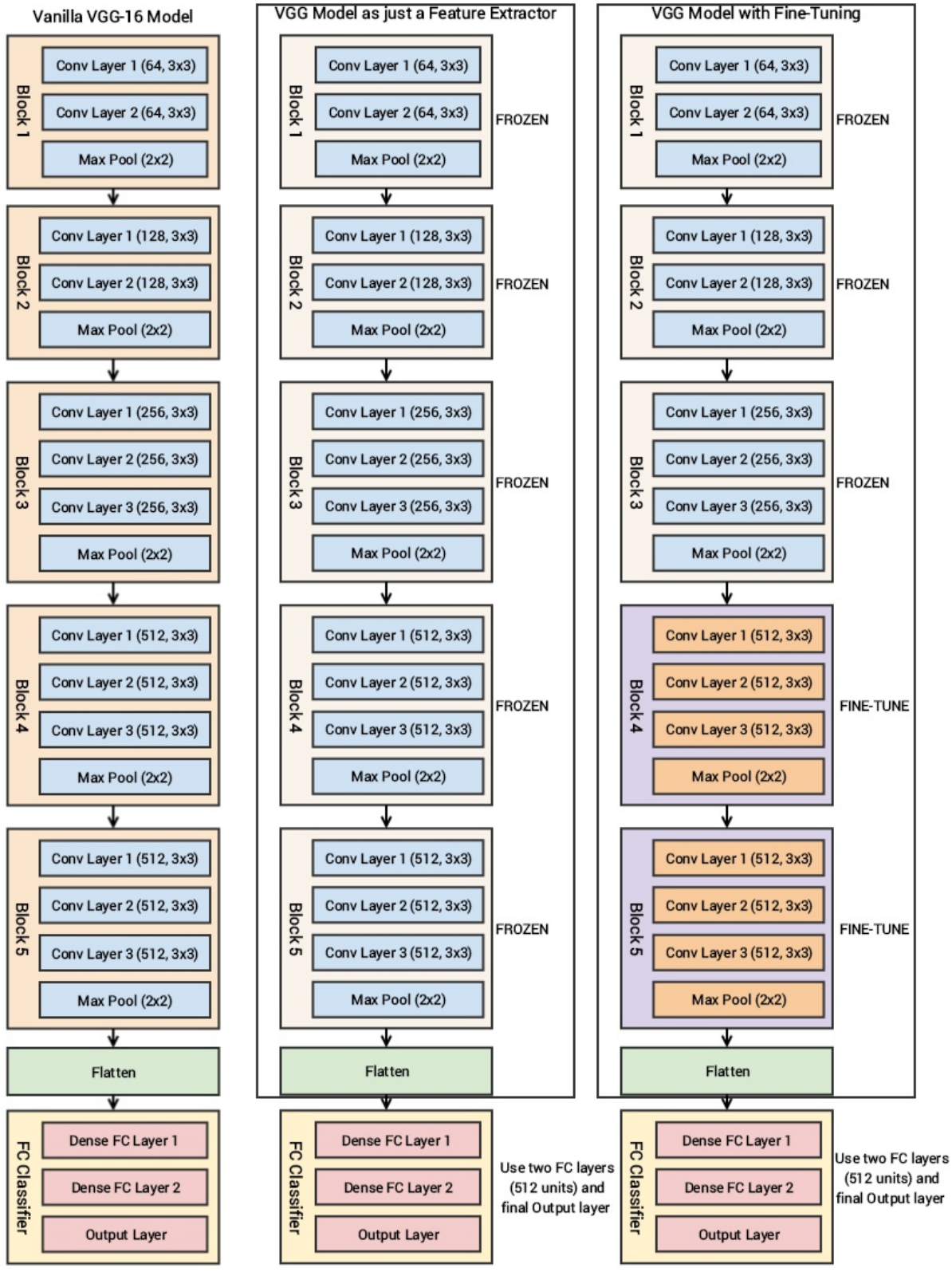
Представим, что есть новый набор данных, и вы хотите научить сеть классифицировать объекты из этой выборки.

- **1. Fine Tuning / дообучение**
  - Берём сеть, обученную на ImageNet.
  - Убираем последние Fully-Connected слои сети, отвечающие за классификацию.
  - Замораживаем веса только нескольких первых слоев сети (`param.requires_grad = False`). Последние веса оставляем обучаемыми.

- Добавляем свои если если нужно. Например, пару FC-слоёв.
- Обучаем получившуюся архитектуру на новых данных.

- **2. Feature Extractor / средство для извлечения признаков**
  - Берём сеть, обученную на ImageNet.
  - Убираем последние Fully-Connected слои сети, отвечающие за классификацию.
  - Замораживаем ( `param.requires_grad = False` ) веса всех предыдущих слоёв.
  - Добавляем свои сли если нужно. Например, пару FC-слоёв.
  - Обучаем на выходах полученной сети свой классификатор (пару FC-слоёв, например) на новых данных.

Ниже эти подходы изображены на примере VGG архитектуры:



В зависимости от нового датасета имеет смысл использовать разные стратегии дообучения:

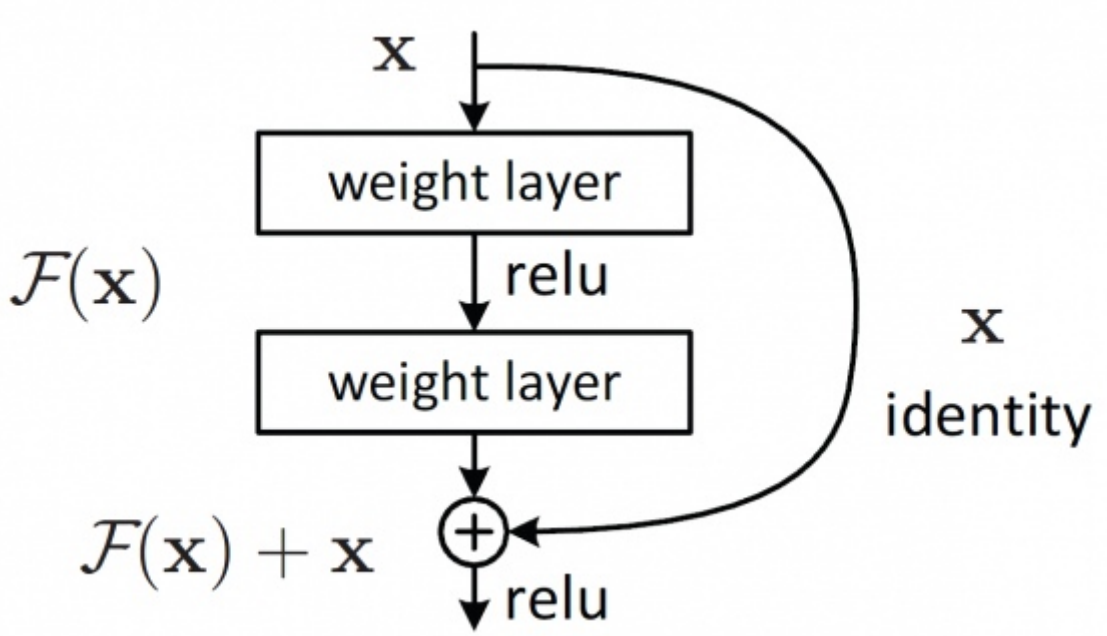
- если датасет *похож* на тот, на котором модель предобучена, то возможно стоит просто заменить слои классификации;
- если датасет *не похож*, то возможно стоит разморозить и сверточные слои тоже.

**Эмпирическое правило**: *чем больше новый датасет не похож* на тот, на котором обучали модель, *тем больше слоев с конца стоит размораживать*.

Если новый датасет достаточно большой (на каждый класс > 1000 изображений), то можно попробовать разморозить всю нейросеть и обучить со случайных весов, как мы это делали до того, как узнали про Transfer Learning.

Рассмотрим **ResNet50** (https://arxiv.org/abs/1512.03385), предобученную на одном из самых крупных датасетов картинок ImageNet, который содержит 1000 классов. Подробнее про данный датасет можно почитать здесь (http://image-net.org/%7D).

Архитектура **ResNet50** основана на residual connections, которые позволяют избежать затухания градиентов:




## 5.2 Изучение модели

Загрузим предобученную модель.

In [28]:

```
from torchvision.models import resnet50

model = resnet50(pretrained=True).to(device)  # скачиваем предобученные веса
```

```
Downloading: "https://download.pytorch.org/models/resnet50-0676ba61.pt
h" to /root/.cache/torch/hub/checkpoints/resnet50-0676ba61.pth

  0%|          | 0.00/97.8M [00:00<?, ?B/s]
```

In [29]:

```
1 model
```

Out[29]:

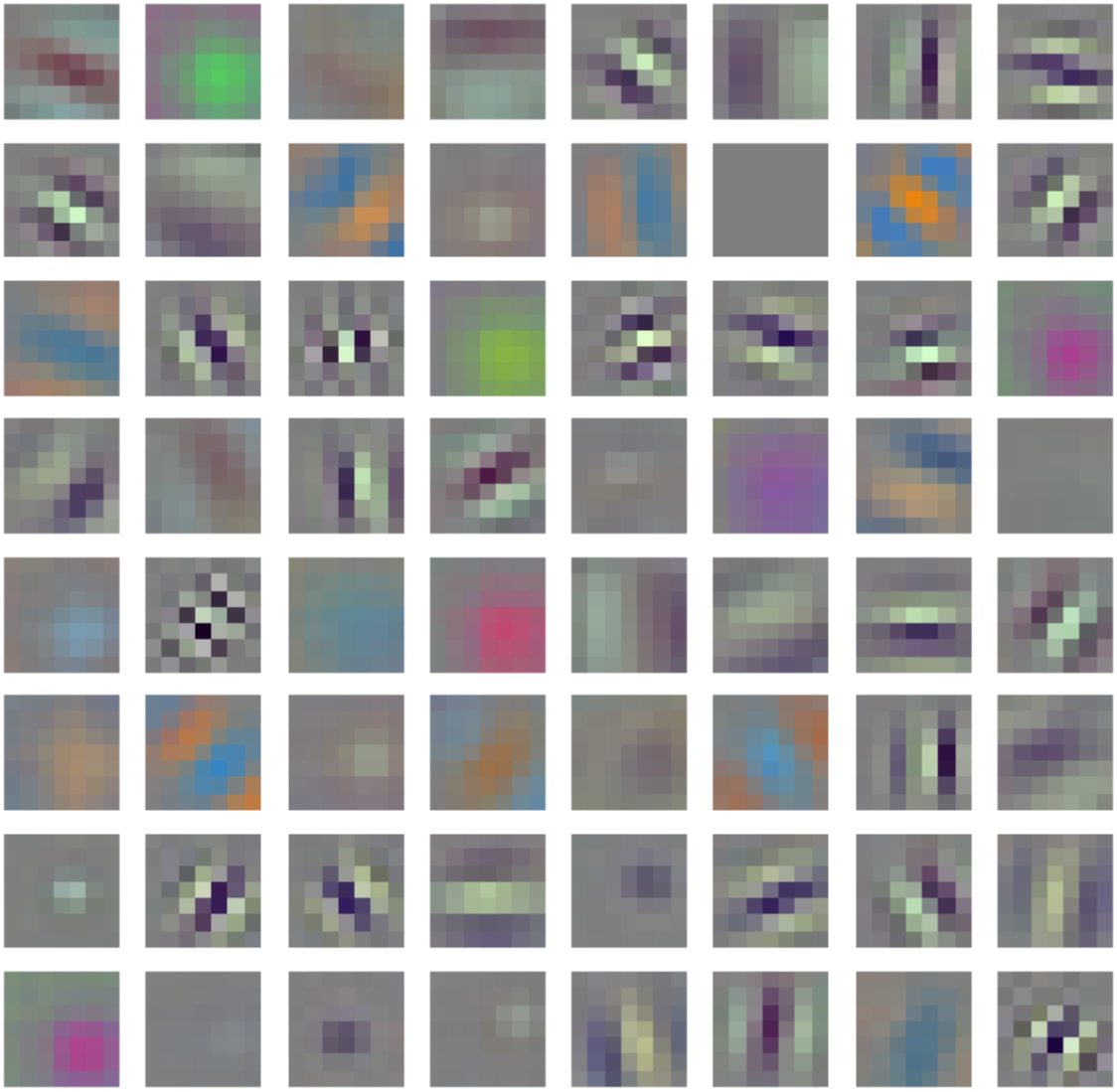
```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=
(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=
1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, t
rack_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), pad
ding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, t
rack_running_stats=True)
```

Посмотрим на веса сверток на первом слое. Так модель предобучена, то они веса уже имеют определенную структуру.

In [30]:

```python
# Минимальные и максимальные значения весов в слое.
max_val = model.conv1.weight.max().detach()
min_val = model.conv1.weight.min().detach()

# Нормировка весов для корректного отображения
weight = (model.conv1.weight - min_val) / (max_val - min_val)
print(f"Веса первого слоя ResNet50. Размер слоя: {weight.shape}")

plt.figure(figsize=(10, 10))
for i, filter in enumerate(weight):
    plt.subplot(8, 8, i + 1)
    plt.imshow(filter.permute(1, 2, 0).detach().cpu())
    plt.axis("off")
```

```
Веса первого слоя ResNet50. Размер слоя: torch.Size([64, 3, 7, 7])
```



Скачаем названия классов для картинок из ImageNet.

In [31]:

```
# class labels
LABELS_URL = 'https://s3.amazonaws.com/deep-learning-models/image-models/imagen
labels = {int(key):value[1] for (key, value) in requests.get(LABELS_URL).json()
```

In [32]:

```
labels
```

Out[32]:

```
{0: 'tench',
 1: 'goldfish',
 2: 'great_white_shark',
 3: 'tiger_shark',
 4: 'hammerhead',
 5: 'electric_ray',
 6: 'stingray',
 7: 'cock',
 8: 'hen',
 9: 'ostrich',
 10: 'brambling',
 11: 'goldfinch',
 12: 'house_finch',
 13: 'junco',
 14: 'indigo_bunting',
 15: 'robin',
 16: 'bulbul',
 17: 'jay',
```

Скачаем картинку альбатроса и посмотрим какой ответ дает предобученная сеть.

In [33]:

```
!wget https://i.ibb.co/60RmQ6S/albatross.jpg -O albatross.jpg
```

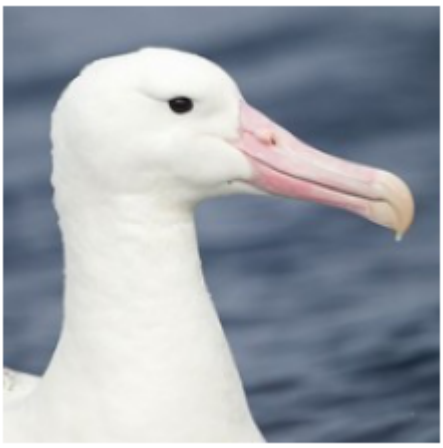
```
--2022-11-29 03:28:15--  https://i.ibb.co/60RmQ6S/albatross.jpg (http
s://i.ibb.co/60RmQ6S/albatross.jpg)
Resolving i.ibb.co (i.ibb.co)... 104.243.38.177, 104.243.38.202
Connecting to i.ibb.co (i.ibb.co)|104.243.38.177|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 11607 (11K) [image/jpeg]
Saving to: ‘albatross.jpg’

albatross.jpg       100%[===================>]  11.33K  --.-KB/s    in
0s

2022-11-29 03:28:16 (83.3 MB/s) - ‘albatross.jpg’ saved [11607/11607]
```

Приведем картинку к нужному формату и напишем функцию для предсказания топ-10 классов для

картинки.

In [34]:

```python
# Приводим изображение к тензору размера 200x200
transform_a = transforms.Compose([
    transforms.ToTensor(),
    transforms.Resize((200, 200)),
])
img = transform_a(plt.imread('albatross.jpg'))

# Покажем получившуюся картинку
plt.imshow(img.permute(1, 2, 0))
plt.axis('off')
plt.show()


def predict(img):
    '''
    Вывести 10 самых вероятных классов, согласно предсказания модели.

    '''
    model.train(False)

    # Добавляем размерность батча, приводим картинку к типу float и переводим на
    img = img.unsqueeze(dim=0).float().to(device)

    # Софтмакс-преобразование логитов нейросети
    probs = torch.nn.functional.softmax(model(img), dim=-1)
    probs = probs.cpu().detach().numpy()

    top_ix = probs.ravel().argsort()[-1:-10:-1]
    print ('top-10 classes are: \n [prob : class label]')
    for l in top_ix:
        print ('%.4f :\t%s' % (probs.ravel()[l], labels[l].split(',')[0]))


predict(img)
```



```
top-10 classes are:
 [prob : class label]
0.9900 :        albatross
0.0042 :        spoonbill
0.0019 :        hammerhead
0.0013 :        American_egret
0.0010 :        goose
0.0002 :        pelican
0.0002 :        oystercatcher
```

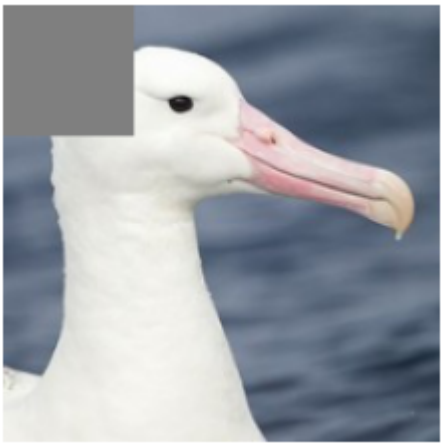
```
0.0001 :        crane
0.0001 :        black_swan
```

Проведем эксперимент по **окклюзии**. Эксперименты по окклюзии проводятся для определения того, какие участки изображений вносят максимамальный вкад в выод нейросети.

Будем каждый раз закрашивать серым цветом квадратный кусочек картинки. Полученную картинку будем передавать в модель. На выходе модели получим вероятность предсказания нужного класса картинки. Используя эти вероятности, заполним heatmap. Таким образом, та область исходной картинки, где вероятность предсказания меньше, является наиболее значимой для модели.

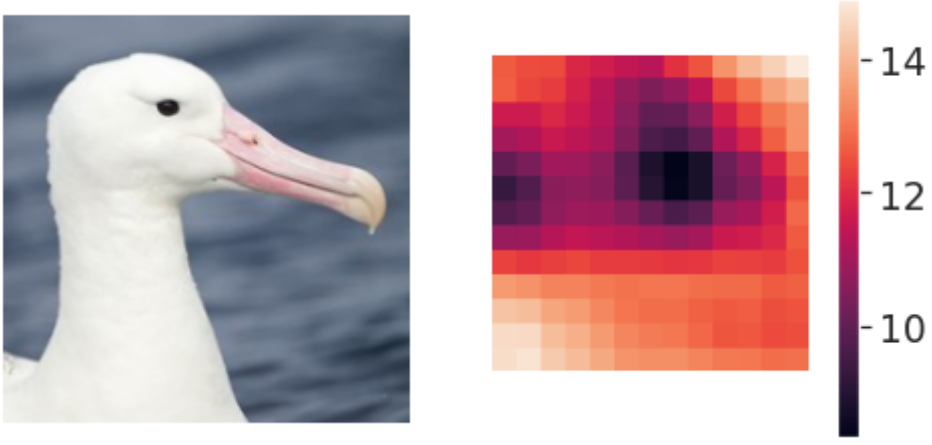
In [35]:

```
# Размер закрашенного кусочка
size = 60

# Так выглядит картинка с закрашенным кусочком
img_new = img.clone()
img_new[:, :size, :size] = 0.5
plt.imshow(img_new.permute(1, 2, 0))
plt.axis('off')
plt.show()
```



In [36]:

```python
# Получим индекс альбатроса среди лейблов
albatross_key = -1
for key, val in labels.items():
    if val.split(", ")[0] == "albatross":
        albatross_key = key

# Шаг между закрашиваемыми квадратиками
stride = size // 4

# Heatmap вероятностей классов
heatmap = torch.zeros((img.shape[1], img.shape[2])).float()
# Нормализующая матрица
norm = torch.zeros((img.shape[1], img.shape[2]))

for i in range(0, 200 - size, stride):
    for j in range(0, 200 - size, stride):
        # Закрашиваем квадратик
        img_new = img.clone()
        img_new[:, i: i + size, j:j + size] = 0.5

        # Получаем вероятность того, что картинка альбатроса
        prob = model(img_new.unsqueeze(0).to(device))[0, albatross_key]

        # Заполняем heatmap и норм. матрицу
        heatmap[i: i + size, j:j + size] += prob.detach().cpu()
        norm[i: i + size, j:j + size] += 1

# Нормализируем heatmap
heatmap = heatmap / norm

# Визуализируем результат
plt.figure(figsize=(8, 4))
# Исходная картинка
plt.subplot(1, 2, 1)
plt.imshow(img.permute(1, 2, 0))
plt.axis('off')
# Heatmap
plt.subplot(1, 2, 2)
plt.imshow(heatmap.detach())
plt.colorbar()
plt.axis('off')
plt.show()
```



Здесь более темные области вносят бОльший вклад в правильную классификацию.

## 5.3 Практика Transfer Learning: Симпсоны

Рассмотрим датасет "Симпсоны". Он скачивается в ячейке ниже, оригинал лежит по ссылке (https://www.kaggle.com/alexattia/the-simpsons-characters-dataset/download).

Чтобы скачать датасет в Google Colab, нужно придерживаться инструкций ниже.

In [37]:

```
# Альтернативно можно раскомментировать эту ячейку, загрузив свой kaggle.json и
# Для того, чтобы скачать kaggle.json, нужно войти в свой аккаунт
# https://www.kaggle.com/<username>/account
# и нажать на "Create New Api Token" (Ctrl+F)

# !pip install -q kaggle
# from google.colab import files

# files.upload()

# ! mkdir ~/.kaggle
# ! cp kaggle.json ~/.kaggle/
# ! chmod 600 ~/.kaggle/kaggle.json
# ! kaggle datasets download -d alexattia/the-simpsons-characters-dataset
# ! unzip the-simpsons-characters-dataset.zip -d <место где будте созранен дата
```

В Kaggle можно добавить этот датасет с пощью кнопки `Add Data` на правой панели в области `Data`. Нужно просто в поиске найти "the-simpsons-characters-dataset" и добавить датасет. Он появится в разделе `Input`.

In [38]:

```
train_dir = '/kaggle/input/the-simpsons-characters-dataset/simpsons_dataset/sim
```

Разделим данные на обучение и валидацию:

In [39]:

```python
class SplitImageFolder():

    def __init__(self, train_dir):

        self.train_dir = train_dir

        # Пути до всех файлов в папке train_dir
        self.train_val_files_path = glob.glob(f'{train_dir}/*/*.jpg')

        # Лейблы для всех файлов в папке train_dir
        self.train_val_labels = [path.split('/')[-2] for path in self.train_val

    def split(self, test_size=0.3):
        # Разделяем файлы на трейн и валидацию
        train_files_path, val_files_path = train_test_split(
            self.train_val_files_path,
            test_size=test_size,
            stratify=self.train_val_labels
        )

        # Сохраняем все трейн и валидацию
        files_path = {'train': train_files_path, 'val': val_files_path}

        return files_path
```

In [40]:

```python
files_path = SplitImageFolder(train_dir).split()
```

Минимальный размер изображения, с которым работает `ResNet50` — $200 \times 200$. В компьютерном зрении часто возникает такая ситуация — картинки в датасете разного размера и качества. Чаще всего их приводят к одному размеру, например, $256 \times 256$ или $512 \times 512$. Приведем все входные изображения к этому размеру с помощью `transforms.Resize` .

In [41]:

```
input_size = 200

# Трансформация / аугментация для обучающих картинок
train_transform = transforms.Compose([
    transforms.Resize(input_size),          # Меняем размер картинки, наименьшая
    transforms.CenterCrop(input_size),      # Вырезаем из центра квадрат размера
    transforms.ColorJitter(0.9, 0.9, 0.9), # Меняем случайно цвета
    transforms.RandomAffine(5),             # Приеменяем слйчайное афинное преоб
    transforms.ToTensor(),                  # Приводим к тензору
])

# Трансформация для валидации
val_transform = transforms.Compose([
    transforms.Resize(input_size),          # Меняем размер картинки, наименьшая
    transforms.CenterCrop(input_size),      # Вырезаем из центра квадрат размера
    transforms.ToTensor(),                  # Приводим к тензору
])


train_dataset = torchvision.datasets.ImageFolder(
    train_dir,
    transform=train_transform,
    is_valid_file=lambda x: x in files_path['train'],
)

val_dataset = torchvision.datasets.ImageFolder(
    train_dir,
    transform=val_transform,
    is_valid_file=lambda x: x in files_path['val']
)
```

In [42]:

```
print("Количество классов: ", len(train_dataset.classes))
```

```
Количество классов:  42
```

Визуализируем данные:

In [43]:

```
# sns.set_style(style='white')

fig, axs = plt.subplots(
    nrows=2, ncols=3, figsize=(16, 12),
    sharey=True, sharex=True
)

for ax in axs.flatten():
    idx = np.random.randint(low=0, high=6000)
    img, label = val_dataset[idx]
    ax.set_title(val_dataset.classes[label])
    ax.grid(b=0)
    ax.set_xticks([])
    ax.set_yticks([])
    ax.imshow(img.numpy().transpose((1, 2, 0)))
```

```
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:12: Matpl
otlibDeprecationWarning: The 'b' parameter of grid() has been renamed
'visible' since Matplotlib 3.5; support for the old name will be dropp
ed two minor releases later.
  if sys.path[0] == "":
```

Инициализируем даталоадеры:

In [44]:

```
batch_size = 64

train_batch_gen = torch.utils.data.DataLoader(train_dataset, batch_size=batch_s
val_batch_gen = torch.utils.data.DataLoader(val_dataset, batch_size=batch_size,
```

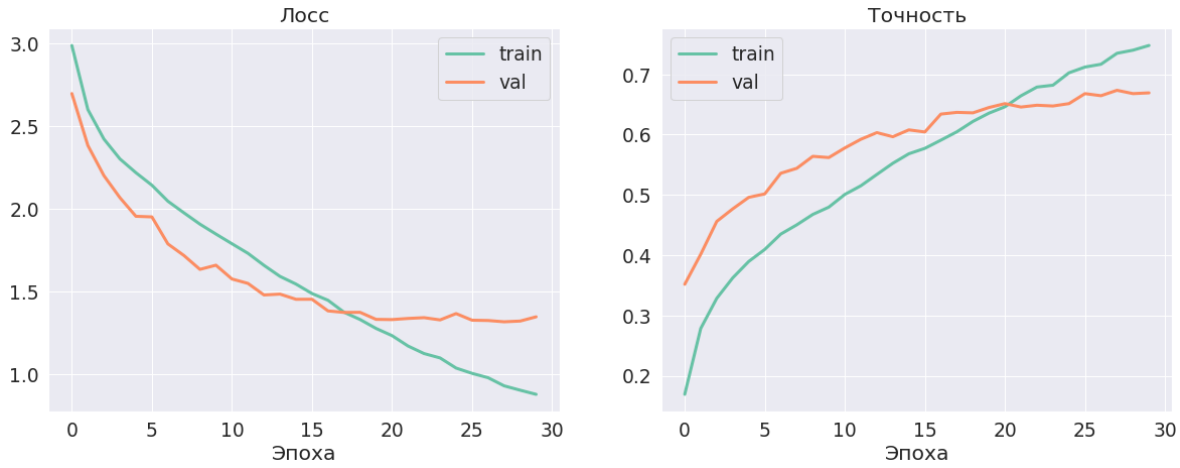
#### 5.3.1. Обучение своей нейросети

Обучим сверточную нейросеть из предыдущей части ноутбука на новых данных.

In [45]:

```
simple_model = SimpleConvNet()

# Нужно заменить FC слой после Flatten, так как размер входного изображения ста
simple_model.fc3 = nn.Linear(147456, 512)
# Нужно заменить последний FC слой, так как количество классов изменилось.
simple_model.fc4 = nn.Linear(512, 47)
simple_model = simple_model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(simple_model.parameters(), lr=0.01)

simple_model, history = train(
    simple_model, criterion, optimizer,
    train_batch_gen, val_batch_gen,
    num_epochs=30,
)
```

```
Epoch 30 of 30 took 208.983s
  training loss (in-iteration):         0.879048
  validation loss (in-iteration):       1.347299
  training accuracy:                    74.78 %
  validation accuracy:                  66.92 %
```




### 5.3.2. Fine Tuning сети ResNet50

Снова инициализируем даталоадеры, так как они являются генераторами:

In [46]:

```
batch_size = 32

train_batch_gen = torch.utils.data.DataLoader(train_dataset, batch_size=batch_s
val_batch_gen = torch.utils.data.DataLoader(val_dataset, batch_size=batch_size,
```

Добавляем новые слои классификации под датасет Симпсонов (47 классов):

In [47]:

```
fine_tuning_model = nn.Sequential()

# предобученная на датасете ImageNet нейросеть ResNet50
fine_tuning_model.add_module('resnet', resnet50(pretrained=True))

# добавляем 2 FC слоя после выходов предобученной неросети
fine_tuning_model.add_module('relu_1', nn.ReLU())
fine_tuning_model.add_module('fc_1', nn.Linear(1000, 512))
fine_tuning_model.add_module('relu_2', nn.ReLU())
fine_tuning_model.add_module('fc_2', nn.Linear(512, 47))

fine_tuning_model = fine_tuning_model.to(device)
```

Убедимся, что все параметры сети "разморожены", то есть являются обучаемыми:

In [48]:

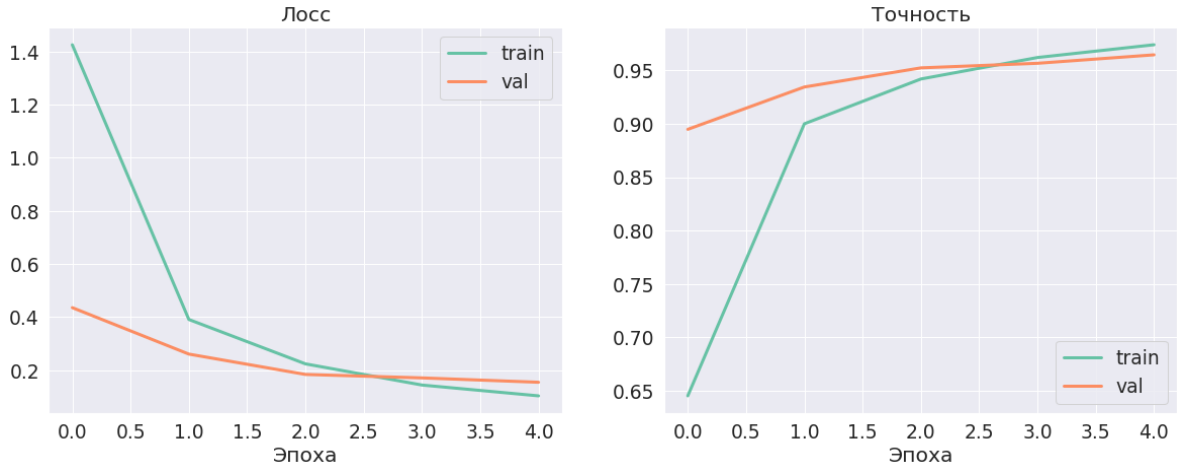
```
for param in fine_tuning_model.parameters():
    assert(param.requires_grad)
    assert(param.is_cuda)
```

Зафайнтюним эту модель на наших данных:

In [49]:

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(fine_tuning_model.parameters(), lr=0.01)

fine_tuning_model, history = train(
    fine_tuning_model, criterion, optimizer,
    train_batch_gen, val_batch_gen,
    num_epochs=5
)
```

```
Epoch 5 of 5 took 218.883s
  training loss (in-iteration):         0.103048
  validation loss (in-iteration):       0.154284
  training accuracy:                    97.41 %
  validation accuracy:                  96.46 %
```



Сравните результаты и сделайте вывод.

### 5.3.3. Feature Extractor сети ResNet50

Заменяем последний слой классификатора на линейный классификатор, заморозим все остальные слои:

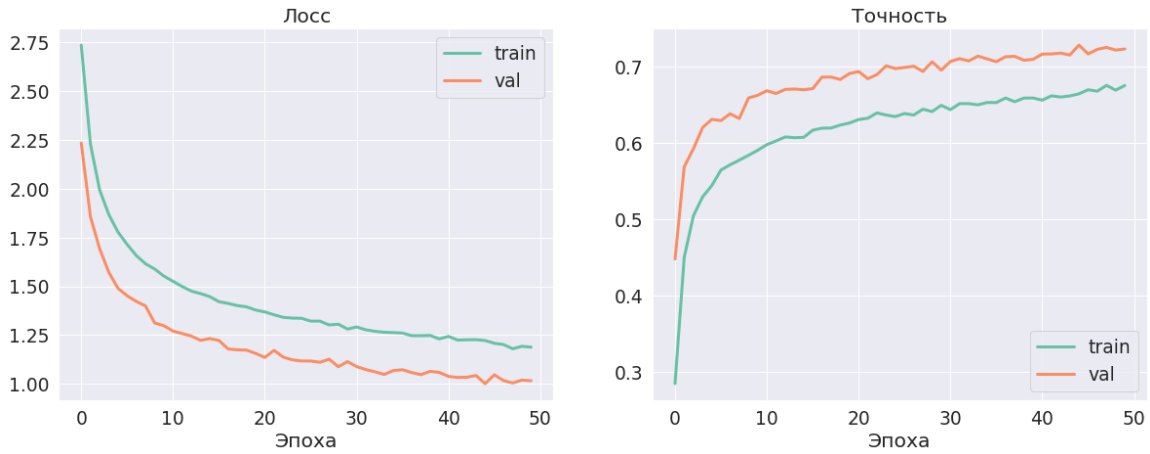
In [50]:

```
clf_model = resnet50(pretrained=True)

# "замораживаем" все веса всех слоев
for param in clf_model.parameters():
    param.requires_grad = False

# этот слой будет обучаемым
clf_model.fc = nn.Linear(2048, 47)
clf_model = clf_model.to(device)
```

In [51]:

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(clf_model.parameters(), lr=0.01)

clf_model, history = train(
    clf_model, criterion, optimizer,
    train_batch_gen, val_batch_gen,
    num_epochs=50
)
```

```
Epoch 50 of 50 took 177.514s
  training loss (in-iteration):         1.188437
  validation loss (in-iteration):       1.016324
  training accuracy:                    67.52 %
  validation accuracy:                  72.30 %
```




Сравните результаты Fine Tuning и Feature Extractor способов и сделайте выводы.

# 6. Нейросетевые дескрипторы

С помощью нейросете можно получить признаковое представление картинки. Это используется для часто для сравнения сгенерированных изображений и реальных изображений. Например на этом основана метрика FID, о ней можно почитать на вики (https://en.wikipedia.org/wiki/Fr%C3%A9chet_inception_distance) или здесь (https://jonathan-hui.medium.com/gan-how-to-measure-gan-performance-64b988c47732). Также признаковое представление можно использовать для других моделей. Например, для генерации текста по картинке в качестве входа в сеть можно использовать не саму картинку, а ее признаковое представление.

Для того, чтобы получить признаковое представление для обучения модели, нужно заменить последний слой классификатора на слой, который ничего не делает. Это можно сделать, создав свой класс Identitiy, а можно использовать nn.Sequential.

То есть оставить тензор признаков как выход нейросети:

In [52]:

```
class Identity(torch.nn.Module):
    def __init__(self):
        super(Identity, self).__init__()

    def forward(self, x):
        return x
```

In [53]:

```
extractor_model = resnet50(pretrained=True)
extractor_model.fc = Identity()

extractor_model.train(False)
```

Out[53]:

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=
(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=
1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, t
rack_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), pad
ding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, t
rack_running_stats=True)
```

Снова инициализируем даталоадеры:

In [54]:

```
batch_size = 1

train_batch_gen = torch.utils.data.DataLoader(train_dataset, batch_size=batch_s
val_batch_gen = torch.utils.data.DataLoader(val_dataset, batch_size=batch_size,
```

Берем нейросетевые признаки с последнего слоя, сохраняя их в переменную X :

In [55]:

```
X = []
Y = []

# То же самое, что и `extractor_model.eval()`
extractor_model.train(False)

# Извлекаем признаки из обучающей выборки
for i, (image_batch, label_batch) in tqdm(enumerate(train_batch_gen),
                                          total=len(train_dataset)/batch_size):
    features = extractor_model(image_batch).detach()
    X.append(features)
    Y.append(label_batch)
    if i == 100:
        break

# Извлекаем признаки из валидационной выборки
for i, (image_batch, label_batch) in tqdm(enumerate(val_batch_gen),
                                          total=len(val_dataset)/batch_size):
    features = extractor_model(image_batch).detach()
    X.append(features)
    Y.append(label_batch)
    if i == 10:
        break
```

```
  0%|          | 0/14653.0 [00:00<?, ?it/s]

  0%|          | 0/6280.0 [00:00<?, ?it/s]
```

In [56]:

```
Y = np.array(Y)
X = np.concatenate(X)

print(X.shape, Y.shape)
```

```
(112, 2048) (112,)

/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:1: Future
Warning: The input object of type 'Tensor' is an array-like implementi
ng one of the corresponding protocols (`__array__`, `__array_interface
__` or `__array_struct__`); but not a sequence (or 0-D). In the futur
e, this object will be coerced as if it was first converted using `np.
array(obj)`. To retain the old behaviour, you have to either modify th
e type 'Tensor', or assign to an empty array created with `np.empty(co
rrect_shape, dtype=object)`.
  """Entry point for launching an IPython kernel.
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:1: Visibl
eDeprecationWarning: Creating an ndarray from ragged nested sequences
(which is a list-or-tuple of lists-or-tuples-or ndarrays with differen
t lengths or shapes) is deprecated. If you meant to do this, you must
specify 'dtype=object' when creating the ndarray.
  """Entry point for launching an IPython kernel.
```

Мы получили признаковое описание объектов и теперь можем работать с ними как с обычным датасетом, или использовать для подсчета FID и т.п.

---

In [ ]:

```
```