Сверточные сети и Transfer learning

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

- Convolution, Pooling - базовые слои, их гиперпараметры и интуиция использования;
- Построение сверточной нейросети для классификации картинок;
- Использование Transfer Learning для этой же задачи.

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
from torchsummary import summary

import torchvision
from torchvision import transforms

from IPython.display import clear_output
%matplotlib inline

sns.set(font_scale=1.2)
# sns.set_style(style='whitegrid')
device_num = 0
torch.cuda.set_device(device_num)
```

```
/usr/local/lib/python3.6/dist-packages/statsmodels/tools/_testing.p
y:19: FutureWarning: pandas.util.testing is deprecated. Use the func
tions in the public API at pandas.testing instead.
  import pandas.util.testing as tm
```

In [0]:

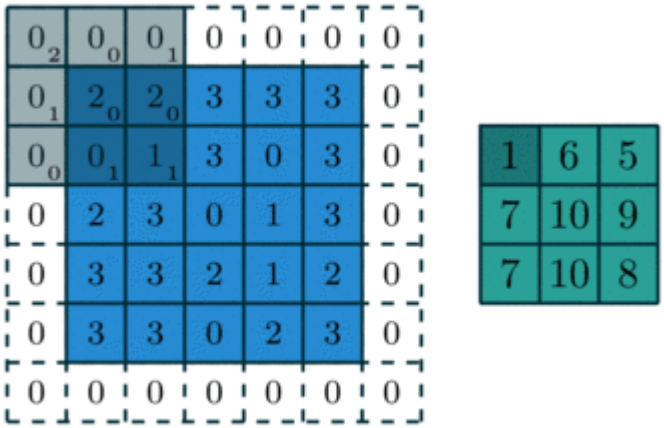
```
device = f"cuda:{device_num}" if torch.cuda.is_available() else "cpu"
```

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.

Задача на понимание:

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

- Какой будет размер выходного изображения?

(224 - 5 + 4) // 2 + 1 = 111 + 1 = 112

= (64, 112, 112)

- Сколько у данного слоя обучаемых параметров?

5 5 3 * 64 + 64

Проверяем себя:

In [6]:

```
5*5*3*64
```

Out[6]:

```
4800
```

In [5]:

```
model = nn.Sequential()
model.add_module('conv1', nn.Conv2d(in_channels=3, out_channels=64,
                                     kernel_size=5, stride=2, padding=2))

from torchsummary import summary

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

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 112, 112]           4,864
================================================================
Total params: 4,864
Trainable params: 4,864
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 6.12
Params size (MB): 0.02
Estimated Total Size (MB): 6.72
----------------------------------------------------------------
```

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

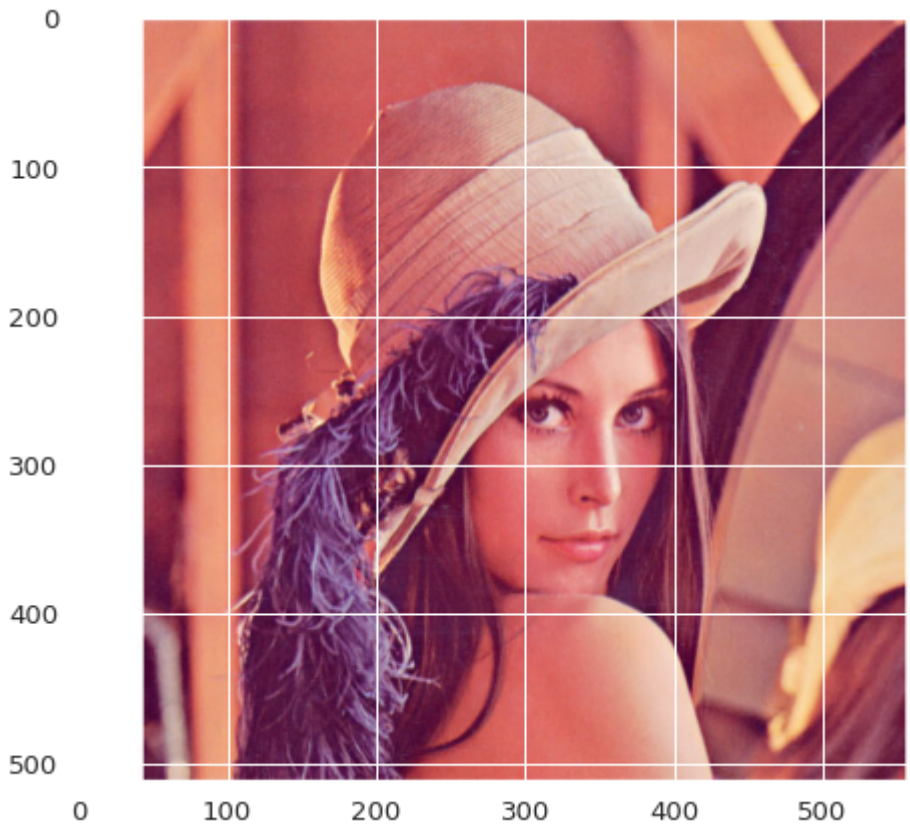
In [7]:

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

```
--2020-05-11 16:21:00--  https://www.kotzendes-einhorn.de/blog/wp-co
ntent/uploads/2011/01/lenna.jpg
Resolving www.kotzendes-einhorn.de (www.kotzendes-einhorn.de)... 94.
130.145.107
Connecting to www.kotzendes-einhorn.de (www.kotzendes-einhorn.de)|9
4.130.145.107|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 64098 (63K) [image/jpeg]
Saving to: ‘lenna.jpg’

lenna.jpg           100%[===================>]  62.60K  --.-KB/s
in 0.1s

2020-05-11 16:21:01 (435 KB/s) - ‘lenna.jpg’ saved [64098/64098]
```

In [8]:

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

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

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



In [13]:

```
type(img)
```

Out[13]:

```
numpy.ndarray
```

Функция для инициализации весов слоя:

In [0]:

```
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 [0]:

```
def convolution(kernel, img, transforms):
    conv = init_conv(kernel)
    img_tensor = transform(img)
    # добавим батч-размерность
    res = conv(img_tensor.unsqueeze(0))
    res = res.detach().squeeze().numpy()
    # пиксели имеют значения от 0 до 255
    res = (np.clip(res, 0, 1)*255).astype(int)
    return res
```

Numpy-картинку нужно привести к torch-тензору:

In [0]:

```
transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Grayscale(),
    transforms.ToTensor()
])
```

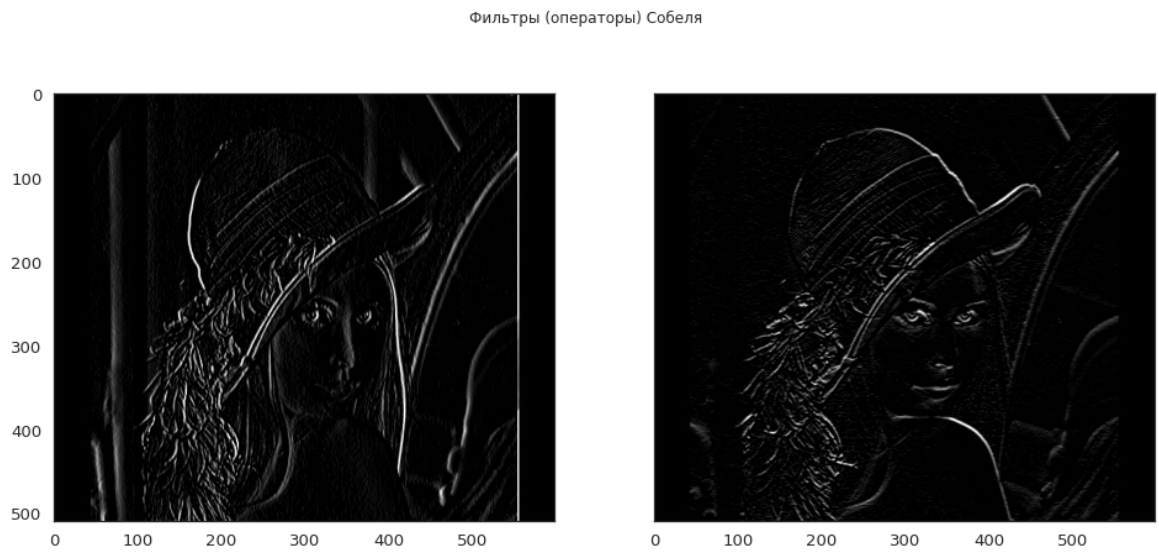
Зададим фильтры:

In [0]:

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

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

In [18]:

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

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

res_images = []
for fig_x, kernel in zip(ax.flatten(), [kernel_1, kernel_2]):
    res = convolution(kernel, img, transform)
    fig_x.imshow(res, cmap='gray')
    res_images.append(res)
plt.suptitle('Фильтры (операторы) Собеля')
plt.show()
```



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 [0]:

```
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.show()
```

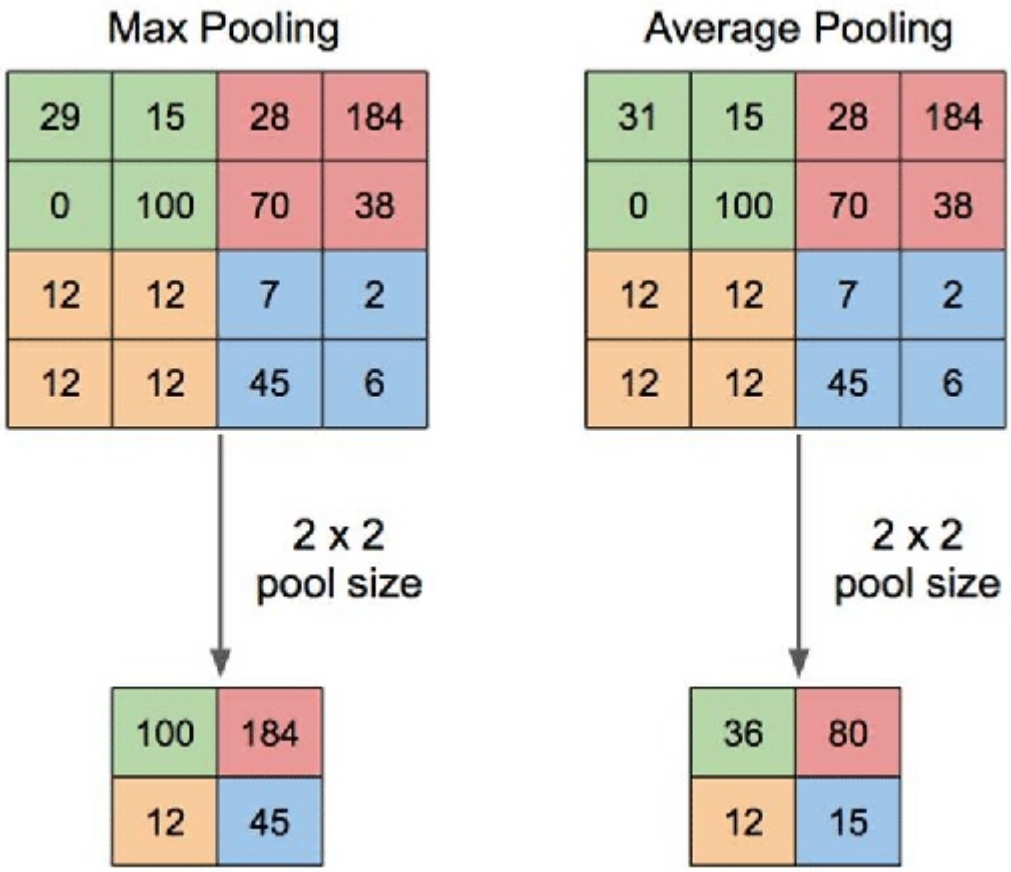
Этот метод называет также [оператором Собеля](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_
(https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_
Так, мы посмотрели как работает свертка на примере оператора Собеля.

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:

1. снижаем размерность изображения и, как следствие, вычислительную сложность
2. 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) 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)

Вопрос: сколько параметров у pooling слоя?

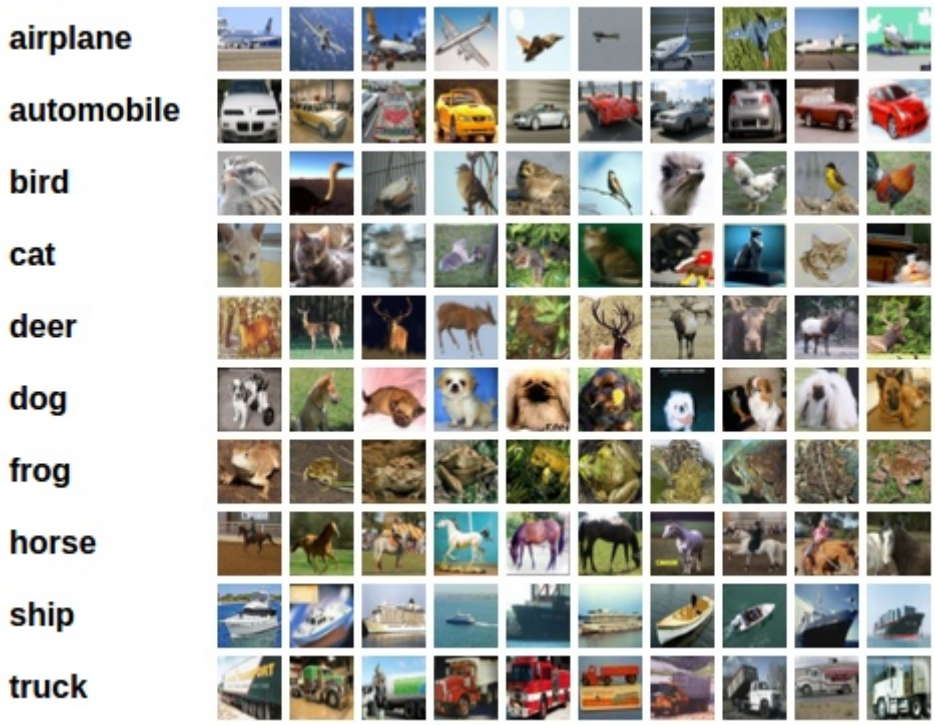
Применим наши знания для решения конкретной задачи:

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 [19]:

```
dataset = torchvision.datasets.CIFAR10(
    root='./data',
    train=True,
    download=True,
    transform=transforms.ToTensor()
)
# разделили выборку на обучение и валидацию
train_dataset, val_dataset = torch.utils.data.random_split(dataset, [40000, 1000
0])

test_dataset = torchvision.datasets.CIFAR10(
    root='./data',
    train=False,
    download=True,
    transform=transforms.ToTensor()
)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
to ./data/cifar-10-python.tar.gz


Extracting ./data/cifar-10-python.tar.gz to ./data

Files already downloaded and verified
```

Инициализируем генераторы батчей:

In [0]:

```
batch_size = 64

train_batch_gen = torch.utils.data.DataLoader(train_dataset, batch_size=batch_si
ze, shuffle=True)
val_batch_gen = torch.utils.data.DataLoader(val_dataset, batch_size=batch_size,
shuffle=True)
test_batch_gen = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size
, shuffle=False)
```

Пайплайн обучения:

In [0]:

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

    :param history: (dict)
        accuracy и loss на обучении и валидации
    '''
    # sns.set_style(style='whitegrid')
    fig = plt.figure(figsize=(20, 7))

    plt.subplot(1,2,1)
    plt.title('Лосс', fontsize=15)
    plt.plot(history['loss']['train'], label='train')
    plt.plot(history['loss']['val'], label='val')
    plt.ylabel('лосс', fontsize=15)
    plt.xlabel('эпоха', fontsize=15)
    plt.legend()

    plt.subplot(1,2,2)
    plt.title('Точность', fontsize=15)
    plt.plot(history['acc']['train'], label='train')
    plt.plot(history['acc']['val'], label='val')
    plt.ylabel('лосс', fontsize=15)
    plt.xlabel('эпоха', fontsize=15)
    plt.legend()
    plt.show()
```

Функция для обучения нейросети:

In [0]:

```
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)

            logits = model(X_batch)

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

            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

            train_loss += np.sum(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 сохраням 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)

            logits = model(X_batch)
            loss = criterion(logits, y_batch.long().to(device))
            val_loss += np.sum(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 сохраням 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
```

Baseline

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

In [0]:

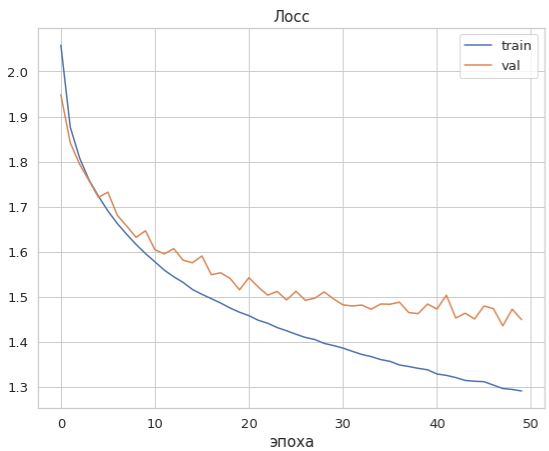
```
class MySimpleModel(nn.Module):
    def __init__(self):
        '''
        Здесь объявляем все слои, которые будем использовать
        '''

        super(MySimpleModel, self).__init__()
        # входное количество признаков = высота * ширина * кол-во каналов картинки
        # 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
```

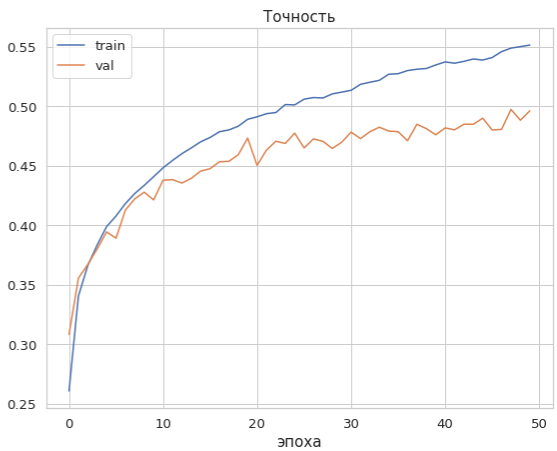
Применим ее к нашим данным — картинками из CIFAR10:

In [0]:

```
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 7.823s
  training loss (in-iteration):         1.290848
  validation loss (in-iteration):       1.449019
  training accuracy:                    55.12 %
  validation accuracy:                  49.60 %
```





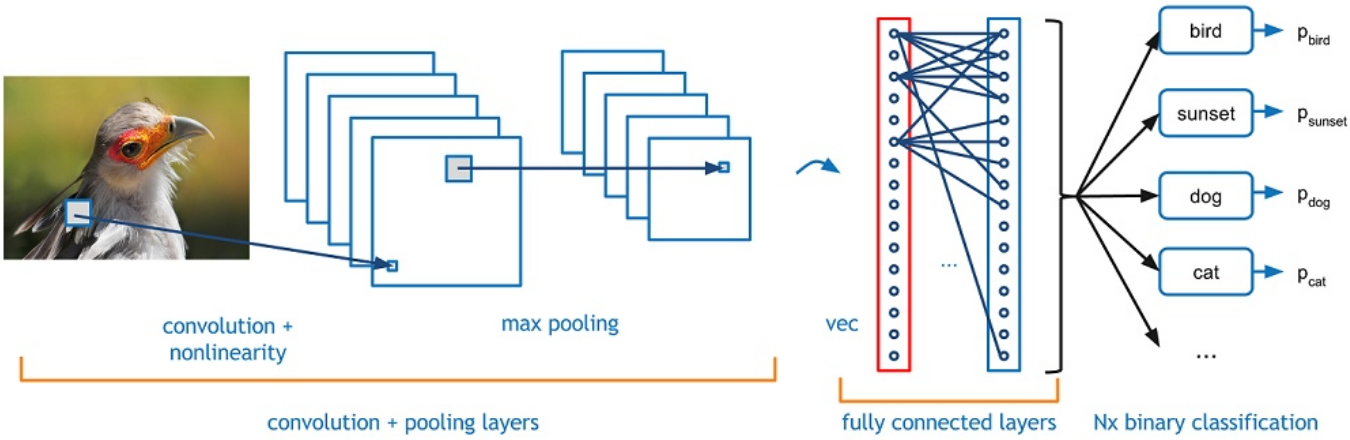
Свёрточная нейросеть

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

Свёрточные нейросети (обыкновенные, есть и намного более продвинутые) 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 (сейчас рассматривается задача классификации изображений).

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 переобучением

Пример, почему 3 свертки 3x3 экономнее по памяти, чем одна свертка 7x7:

Предположим, что входное изображение имеет C каналов, и что количество фильтров в сверточных слоях тоже равно C.

1). **[CONV((3, 3)) -> RELU]^3**

Количество параметров: $3(C(C * 3 * 3)) = 27C^2$

Рецептивное поле первой свертки на входном изображении: 3×3

Рецептивное поле второй свертки на выходе первой свертки: 3×3, и, следовательно, 5×5 на входном изображении.

Аналогично рецептивное поле третьей свертки (а значит и всего блока) на входном изображении: 7×7

2). **[CONV((7, 7)) -> RELU]^1**

Количество параметров: $C(C * 7 * 7) = 49C^2$

Рецептивное поле на входном изображении: 7×7

Вывод: Первый вариант формирует более сложные признаки из-за нелинейностей, при этом имеет меньше параметров и такое же рецептивное поле на входном изображении.

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.

Задание:

Посмотрите на следующую нейросеть и укажите на некорректные шаги в реализации.

In [0]:

```
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)

Задание:

На основе предыдущего примера:

1. Исправьте все ошибки и реализуйте свою сверточную сеть;
2. Обучите её и посмотрите на качество.

Не забывайте о пользе `Dropout` и `BatchNorm`.

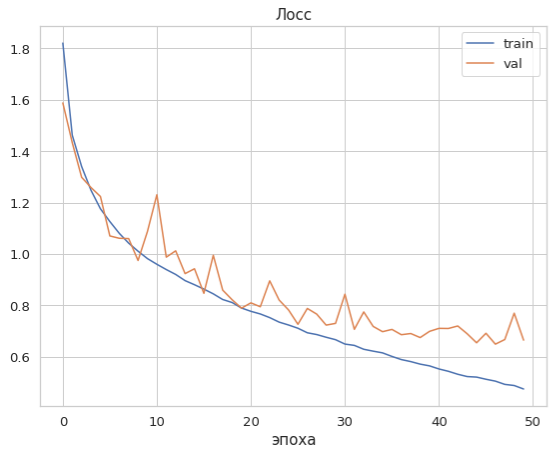
In [0]:

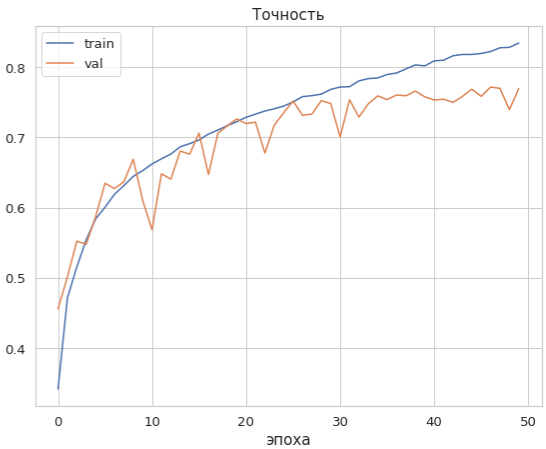
```
class SimpleConvNet(nn.Module):
    def __init__(self):
        # вызов конструктора предка
        super(SimpleConvNet, self).__init__()

#           <Ваш код здесь>

    def forward(self, x):

#           <Ваш код здесь>

        return out
```

In [0]:

```
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
)
```

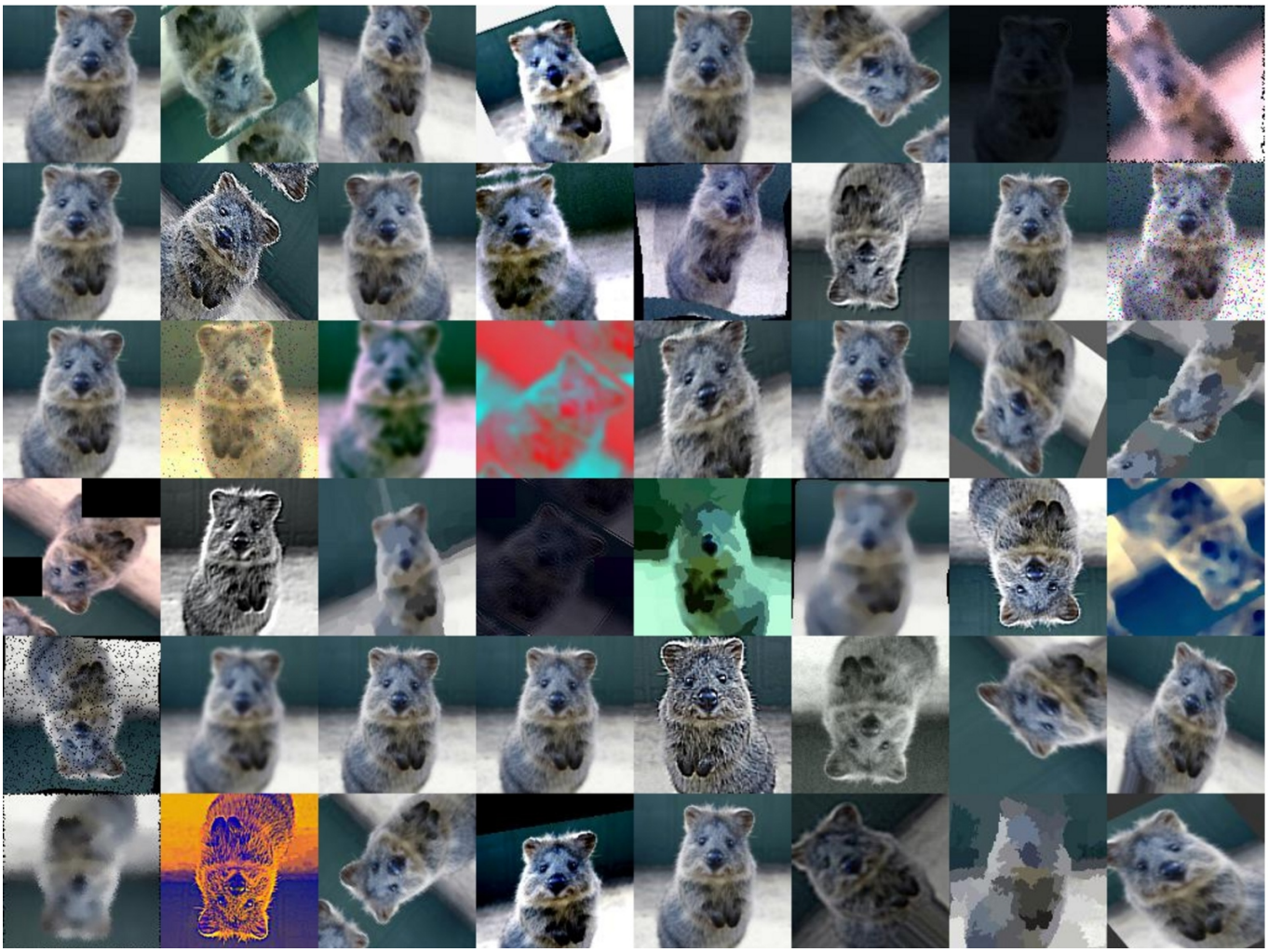
```
Epoch 50 of 50 took 9.164s
  training loss (in-iteration):         0.474847
  validation loss (in-iteration):       0.664767
  training accuracy:                    83.43 %
  validation accuracy:                  76.95 %
```





> Сравните полученное качество с тем, что мы получили ранее для MLP.

Data augmentations



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".

С помощью `torchvision.transforms` мы применим несколько случайных преобразований к картинкам и тем самым расширим нашу выборку. Про все реализованные в библиотеке преобразования можно почитать здесь (https://pytorch.org/docs/stable/torchvision/transforms.html).

В данном случае в качестве аугментации мы используем:

- `ColorJitter` - изменение яркости, контраста, насыщенности цветов
- `RandomAffine` - афинное преобразование

In [0]:

```
from torchvision import transforms


# набор аугментаций при обучении
transform_train = transforms.Compose([
    transforms.ColorJitter(0.9, 0.9, 0.9),
    transforms.RandomAffine(5),
    transforms.ToTensor(),
])

# набор аугментаций при валидации
transform_test = transforms.Compose([
    transforms.ToTensor(),
])
```

Загружаем данные, но уже *с аугментациями*:

In [0]:

```
dataset = torchvision.datasets.CIFAR10(
    root='./data',
    train=True,
    download=True,
    transform=transform_train
)
# разделили выборку на обучение и валидацию
train_dataset, val_dataset = torch.utils.data.random_split(dataset, [40000, 1000
0])

test_dataset = torchvision.datasets.CIFAR10(
    root='./data',
    train=False,
    download=True,
    transform=transform_test
)
```

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

Инициализируем даталоадеры:

In [0]:

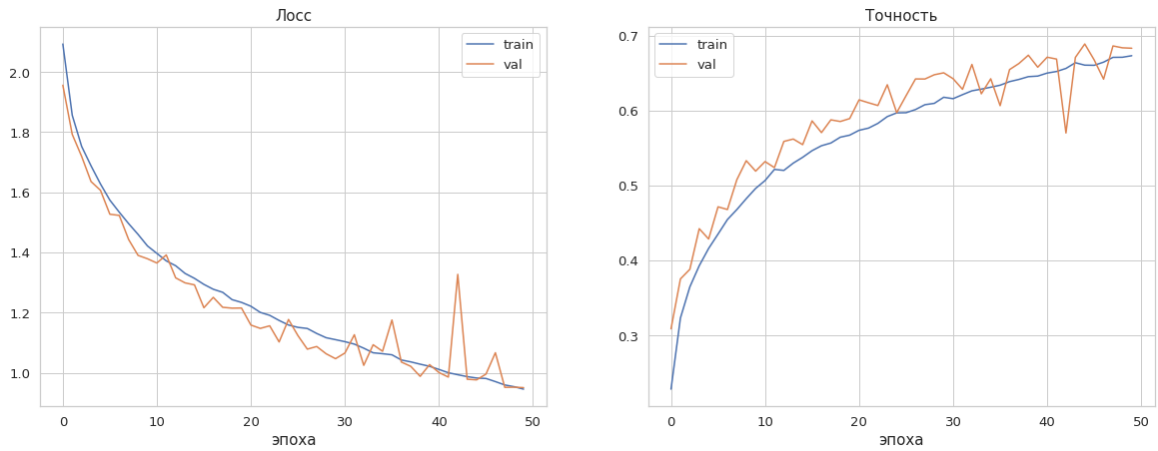
```
batch_size = 64

train_batch_gen = torch.utils.data.DataLoader(train_dataset, batch_size=batch_si
ze, shuffle=True)
val_batch_gen = torch.utils.data.DataLoader(val_dataset, batch_size=batch_size,
shuffle=True)
test_batch_gen = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size
, shuffle=False)
```

Обучим нейросеть на новых данных:

In [0]:

```
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 24.699s
  training loss (in-iteration):         0.945307
  validation loss (in-iteration):       0.950316
  training accuracy:                    67.30 %
  validation accuracy:                  68.29 %
```



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.

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 (миллионы картинок) датасете (например, ImageNet ~ 14 млн картинок).

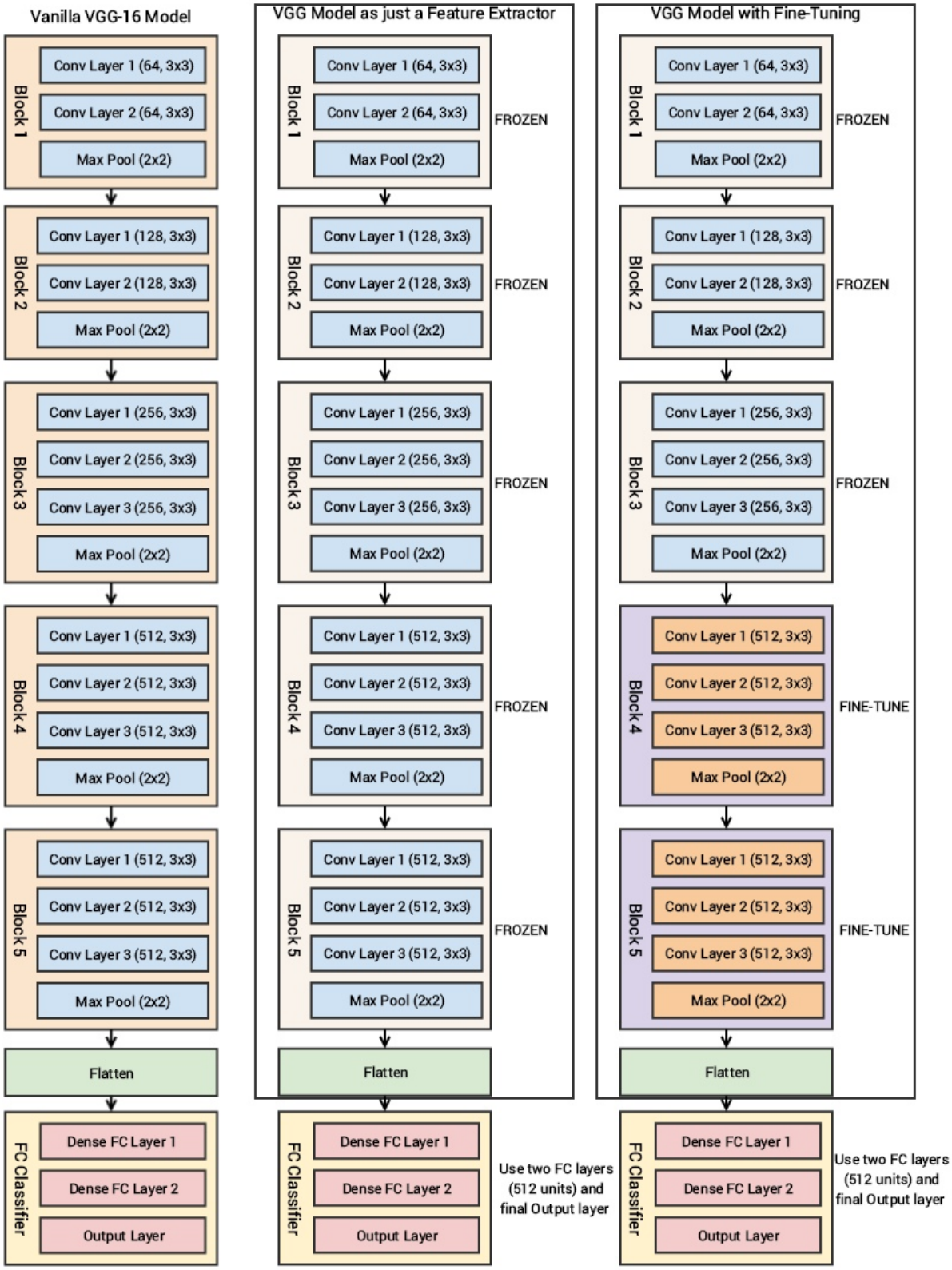
На данный момент есть множество предобученных моделей: `AlexNet`, `DenseNet`, `ResNet`, `VGG`, `Inception` и другие, а также их различные модификации. Все они отличаются архитектурой и входными данными.

Описание метода:

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

- **1. Fine Tuning** (дообучение CNN)
  - Берём сеть, обученную на ImageNet;
  - Убираем последние Fully-Connected слои сети, отвечающие за классификацию;
  - Размораживаем все или несколько предыдущих слоев сети (`param.requires_grad = True`), начиная с последнего (**не** с первого);
  - Обучаем получившуюся архитектуру на новых данных (пару FC-слоёв, например).
- **2. Feature Extractor** (CNN как 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
  - Обучаем на выходах полученной сети свой классификатор (пару FC-слоёв, например) на новых данных.

Ниже эти подходы изображены на примере VGG архитектуры:


Vanilla VGG-16 Model
VGG Model as just a Feature Extractor
VGG Model with Fine-Tuning
Block 1
Conv Layer 1 (64, 3x3)
Conv Layer 2 (64, 3x3)
Max Pool (2x2)
Block 2
Conv Layer 1 (128, 3x3)
Conv Layer 2 (128, 3x3)
Max Pool (2x2)
Block 3
Conv Layer 1 (256, 3x3)
Conv Layer 2 (256, 3x3)
Conv Layer 3 (256, 3x3)
Max Pool (2x2)
Block 4
Conv Layer 1 (512, 3x3)
Conv Layer 2 (512, 3x3)
Conv Layer 3 (512, 3x3)
Max Pool (2x2)
Block 5
Conv Layer 1 (512, 3x3)
Conv Layer 2 (512, 3x3)
Conv Layer 3 (512, 3x3)
Max Pool (2x2)
Flatten
FC Classifier
Dense FC Layer 1
Dense FC Layer 2
Output Layer
FROZEN
FINE-TUNE
Use two FC layers (512 units) and final Output layer

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

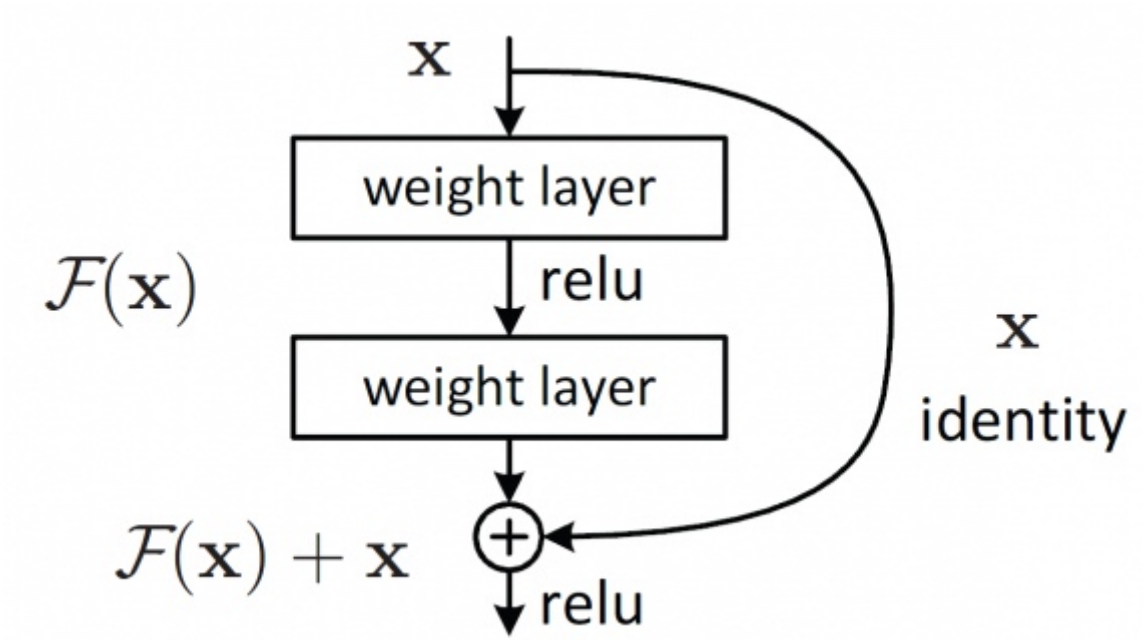
- если датасет *похож* на тот, на котором модель предобучена, то возможно стоит просто заменить слой классификации;
- если датасет *не похож*, то возможно стоит разморозить и сверточные слои тоже.

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/}).

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



Загрузим предобученную модель.

In [0]:

```
from torchvision.models import resnet50

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

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.
pth" to /root/.cache/torch/checkpoints/resnet50-19c8e357.pth
```

Скачаем названия классов для картинок из ImageNet.

In [0]:

```
# class labels
LABELS_URL = 'https://s3.amazonaws.com/outcome-blog/imagenet/labels.json'
labels = {int(key):value for (key, value) in requests.get(LABELS_URL).json().ite
ms()}
```

Скачаем картинку альбатроса и посмотрим какой ответ дает предобученная сеть.

In [0]:

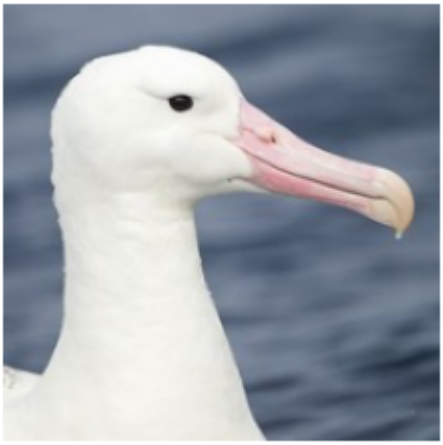
```
!wget https://i.ibb.co/60RmQ6S/albatross.jpg -O albatross.jpg
```

```
--2020-05-10 21:08:03--  https://i.ibb.co/60RmQ6S/albatross.jpg
Resolving i.ibb.co (i.ibb.co)... 206.221.176.29, 199.127.61.88
Connecting to i.ibb.co (i.ibb.co)|206.221.176.29|: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

2020-05-10 21:08:04 (160 MB/s) - ‘albatross.jpg’ saved [11607/11607]
```

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

In [0]:

```
from skimage.transform import resize

# приводим изображение к размеру 200x200
img = resize(plt.imread('albatross.jpg'), (200, 200))

# покажем получившуюся картинку
plt.imshow(img)
plt.axis('off')
plt.show()


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

    '''
    model.train(False)

    # ! Обратите внимание на то, как картинка-`np.array` приводится к картинке-тензору
    img = torch.FloatTensor(img.reshape([1, 200, 200, 3]).transpose([0, 3, 1, 2
]))

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

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).

In [0]:

```
# Альтернативно можно раскомментировать эту ячейку, загрузив свой kaggle.json из Kaggle.
# 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 simpsons_dataset
```

In [0]:

```
import subprocess

def upload_file_from_gdrive(file_id, outfile):
    upload_cmd = (
        "wget --load-cookies /tmp/cookies.txt"
        " \"https://docs.google.com/uc?export=download&confirm=$("
        " wget --quiet --save-cookies /tmp/cookies.txt --keep-session-cookies"
        f" --no-check-certificate 'https://docs.google.com/uc?export=download&id
={file_id}'"
        f" -O- | sed -rn 's/.*confirm=([0-9A-Za-z_]+).*/\\1\\n/p')&id={file_id}
\" "
        f" -O {outfile} && rm -rf /tmp/cookies.txt"
    )
    subprocess.check_call(upload_cmd, shell=True)

upload_file_from_gdrive(
    file_id='1yMJn1db-T7HETlVpKMkvhrcJ-_sT8JMx',
    outfile='the-simpsons-characters-dataset.zip'
)
```

In [0]:

```
!unzip the-simpsons-characters-dataset.zip -d simpsons_dataset
```

In [0]:

```
train_dir = 'simpsons_dataset/simpsons_dataset'
```

Разделим данные на обучение и валидацию:

In [0]:

```
class SplitImageFolder():

    def __init__(self, train_dir):

        self.train_dir = train_dir
        self.train_val_files_path = glob.glob(f'{train_dir}/*/*.jpg')
        self.train_val_labels = [path.split('/')[-2] for path in self.train_val_
files_path]

    def split(self, test_size=0.3):

        train_files_path, val_files_path = train_test_split(
            self.train_val_files_path,
            test_size=test_size,
            stratify=self.train_val_labels
        )

        files_path = {'train': train_files_path, 'val': val_files_path}

        return files_path
```

In [0]:

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

Минимальный размер изображения, с которым работает `ResNet50` — 200×200.

В компьютерном зрении часто возникает такая ситуация -- картинки в датасете разного размера и качества. Чаще всего их приводят к одному размеру, например, 256×256 или 512×512.

Приведем все входные изображения к этому размеру с помощью `transforms.Resize`.

In [0]:

```
input_size = 200

train_transform = transforms.Compose([
    transforms.Resize(input_size),
    transforms.CenterCrop(input_size),
    transforms.ColorJitter(0.9, 0.9, 0.9),
    transforms.RandomAffine(5),
    transforms.ToTensor(),
])


val_transform = transforms.Compose([
    transforms.Resize(input_size),
    transforms.CenterCrop(input_size),
    transforms.ToTensor(),
])

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 [0]:

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

```
Количество классов:  47
```

Визуализируем данные:

In [0]:

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

fig, ax = plt.subplots(
    nrows=2, ncols=3, figsize=(8, 6),
    sharey=True, sharex=True
)

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

Инициализируем даталоадеры:

In [0]:

```
batch_size = 64

train_batch_gen = torch.utils.data.DataLoader(train_dataset, batch_size=batch_si
ze, shuffle=True)
val_batch_gen = torch.utils.data.DataLoader(val_dataset, batch_size=batch_size,
shuffle=False)
```

Обучение своей нейросети

Обучим сверточную нейросеть из предыдущей части задания на новых данных:

In [ ]:

```
simple_model = SimpleConvNet()

# нужно заменить FC слой после Flatten, так как размер входного изображения стал больше
simple_model.fc3 = nn.Linear(147456, 512)
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
)
```

Fine Tuning сети ResNet50

In [0]:

```
# summary(fine_tuning_model.to(device), (3, 200, 200))
```

Снова инициализируем даталоадеры, так как они являются генераторами:

In [0]:

```
batch_size = 32

train_batch_gen = torch.utils.data.DataLoader(train_dataset, batch_size=batch_si
ze, shuffle=True)
val_batch_gen = torch.utils.data.DataLoader(val_dataset, batch_size=batch_size,
shuffle=False)
```

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

In [0]:

```
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 [0]:

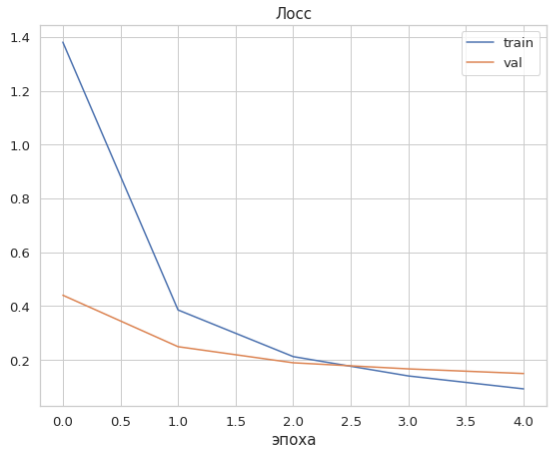
```
for param in fine_tuning_model.parameters():
    assert(param.requires_grad)
    assert(param.is_cuda)
```

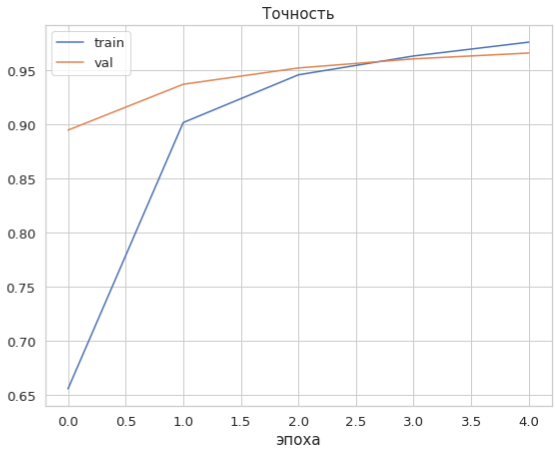
Зафайнтюним эту модель на наших данных:

In [0]:

```
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 446.718s
  training loss (in-iteration):         0.092902
  validation loss (in-iteration):       0.149881
  training accuracy:                    97.63 %
  validation accuracy:                  96.62 %
```





> Сравните результаты и сделайте вывод.

Feature Extractor сети ResNet50

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

In [0]:

```
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 [ ]:

```
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
)
```

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

Задание *

Попробуйте дообучить другие архитектуры:

- `DenseNet`
- `MnasNet`
- `InceptionV3`

In [0]:

```
<Ваш код здесь>
```

Нейросетевые дескрипторы

Для того, чтобы получить признаковое представление для обучения модели, нужно заменить последний слой классификатора на слой, который ничего не делает.

То есть оставить тензор признаков как выход нейросети:

In [0]:

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

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

In [ ]:

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

extractor_model.train(False)
```

Снова инициализируем даталоадеры:

In [0]:

```
batch_size = 1

train_batch_gen = torch.utils.data.DataLoader(train_dataset, batch_size=batch_si
ze, shuffle=True)
val_batch_gen = torch.utils.data.DataLoader(val_dataset, batch_size=batch_size,
shuffle=False)
```

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

In [0]:

```
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
```

In [0]:

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

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

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

Мы получили признаковое описание объектов и теперь можем работать с ними как с обычным датасетом.