PyTorch и Многослойные нейронные сети

In [ ]:

```
from collections import defaultdict
from collections import OrderedDict

import time

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import seaborn as sns
sns.set(palette='Set2', font_scale=1.2)

from IPython.display import clear_output

from sklearn.datasets import load_boston
```

In [ ]:

```
#!pip3 install torch
import torch
from torch import nn
import torch.nn.functional as F

import torchvision
from torchvision import transforms

print(torch.__version__)
```

```
1.10.0+cu111
```

Высокоуровневый pytorch

До этого мы работали с низкоуровневым Pytorch. Строить глубокие нейронные сети таким образом будет долго. В Pytorch-е есть также предопределенные слои, активации и оптимизаторы.

1 Цикл обучения модели: подробнее

Выше мы уже вспомнили, из чего состоит цикл обучения нейросети.

1. **Прямой проход / Forward pass**
   Считаем $\widehat{y}$ и также запоминаем значения выходов всех слоев.

2. **Подсчет функции потерь / Loss calculating**
   Вычисление эмпирического риска на текущем наборе объектов. Эмпирический риск далее будем называть лоссом.

3. **Обратный проход / Backward pass**
   Считаем градиенты по всем обучаемым параметрам и запоминаем их.

4. **Шаг оптимизации / Optimization step**
   Делаем шаг градиентного спуска, обновляя все обучаемые веса.

На **PyTorch** цикл обучения в общем случае выглядит так:

```
for i in range(num_epochs):

    y_pred = model(x)               # forward pass

    loss = loss_function(y_pred, y) # вычисление эмпирического риска (лосс
a)

    loss.backward()                 # backward pass

    optimizer.step()                # шаг оптимизации

    optimizer.zero_grad()           # зануляем градиенты
```

Вспомним, какие есть:

1. **Типы слоев в нейронной сети** (включая функции активации)

- `Linear`
- `ReLU`
- `LeakyReLU`
- `ELU`
- `SoftMax`
- `Dropout`
- `BatchNorm`
- ... ?

2. **Функции потерь**
   Пусть n — размер выборки.

- `LogLoss / BinaryCrossEntropy / BCE` : *бинарная классификация*. Пусть $\widehat{y_i} \in (0, 1), y \in \{0, 1\}$, тогда
$$\mathcal{L}(\widehat{y}, y) = \frac{1}{n} \sum_{i=1}^{n} \left[y_i \log \widehat{y}_i + (1 - y_i) \log (1 - \widehat{y}_i) \right]$$
- `CrossEntropy` : *многоклассовая классификация*. Пусть $\widehat{y}_{ij} \in (0, 1)^n, y_{ij} \in \{0, 1\}^n, K -$ количество классов, тогда
$$\mathcal{L}(\widehat{y}, y) = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{K} y_{ij} \log \widehat{y}_{ij}$$
- `MSELoss` : *регрессия*
  Пусть $\widehat{y}, y \in \mathbb{R}$, тогда
$$\mathcal{L}(\widehat{y}, y) = \frac{1}{n} \sum_{i=1}^{n} (\widehat{y} - y)^2$$
- ... ?

3. **Оптимизаторы**

- `SGD`

- AdaGrad
- RMSProp
- AdaDelta
- Adam
- ... ?

Welcome to Ноутбук по методам оптимизации.

2 Практическая часть

Для ознакомления с высокоуровневым интерфейсом будем решать задачу классификации картинок на 10 классов на датасете CIFAR10 из 60k картинок размера 3x32x32.

Скачаем картинки и посмотрим на них:

In [ ]:

```
!wget https://raw.githubusercontent.com/riknel/ML_lectures/master/cifar.py
```

```
--2022-03-05 16:46:10--  https://raw.githubusercontent.com/riknel/ML_l
ectures/master/cifar.py (https://raw.githubusercontent.com/riknel/ML_l
ectures/master/cifar.py)
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 18
5.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|18
5.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2397 (2.3K) [text/plain]
Saving to: ‘cifar.py’

cifar.py            100%[===================>]   2.34K  --.-KB/s    in
0s

2022-03-05 16:46:10 (57.2 MB/s) - ‘cifar.py’ saved [2397/2397]
```

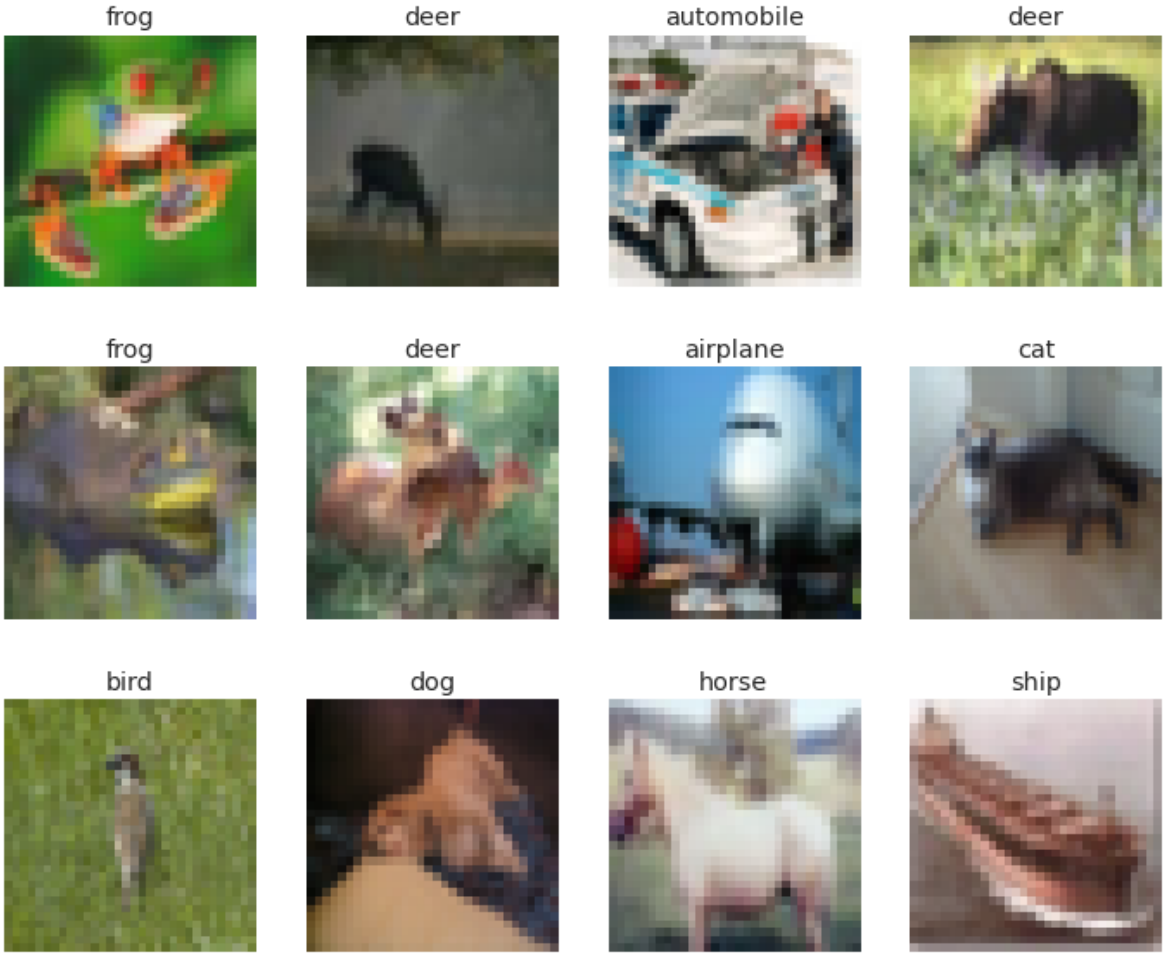
In [ ]:

```
%%time
from cifar import load_cifar10

X_train, y_train, X_val, y_val, X_test, y_test = load_cifar10("cifar_data")

class_names = np.array([
    'airplane', 'automobile', 'bird', 'cat', 'deer',
    'dog', 'frog', 'horse', 'ship', 'truck'
])

print(type(X_train), type(y_train))
print(X_train.shape, y_train.shape)
```

```
Dataset not found. Downloading...
<class 'numpy.ndarray'> <class 'numpy.ndarray'>
(40000, 3, 32, 32) (40000,)
CPU times: user 2.25 s, sys: 1.01 s, total: 3.26 s
Wall time: 4.65 s
```

In [ ]:

```
plt.figure(figsize=(12, 10))
for i in range(12):
    plt.subplot(3, 4, i + 1)
    plt.axis('off')
    plt.title(class_names[y_train[i]])
    plt.imshow(np.transpose(X_train[i],[1, 2, 0]))
```



Обучение проходит по батчам. Поэтому нам нужно либо самим написать генератор для батчей, либо использовать уже написанный за нас класс `DataLoader`. `DataLoader` принимает в аргументах размер батча и также ему можно сказать, нужно ли перемешивать данные. Пока что сами напишем генератор:

In [ ]:

```python
def batch_generator(X, y, batchsize, device, shuffle=True):
    '''
        Генерирует tuple из батча объектов и их меток
        X: np.ndarray -- выборка
        y: np.ndarray -- таргет
        batchsize: int -- размер батча
        device: str -- утсройство, на котором будут производиться вычисления
        shuffle: bool -- перемешивать выборку или нет
    '''

    indices = np.arange(len(X))

    # Во время обучения перемешиваем, во время тестирования - нет
    if shuffle:
        indices = np.random.permutation(indices)

    # Идем по всем данным с шагом batchsize.
    # Возвращаем start: start + batchsize объектов на каждой итерации
    for start in range(0, len(indices), batchsize):
        ix = indices[start: start + batchsize]

        # Переведем массивы в соотв. тензоры.
        # Для удобства переместим выборку на наше устройство (GPU).
        yield torch.FloatTensor(X[ix]).to(device), torch.LongTensor(y[ix]).to(d
```

2.1 Создание модели (Sequential-стиль)

Главной абстракцией в PyTorch является `torch.nn.Module`. По сути модуль можно понимать как нейронную сеть или ее какую-то часть. Каждый стандартный слой в PyTorch-е наследуются от `torch.nn.Module`.

Модуль это нечто, что имеет метод `forward` и, возможно, `backward`. Но вообще `backward` является автоматическим (`autograd`). Кроме того, модуль может содержать в себе другие модули.

In [ ]:

```
1 print(nn.Module.__doc__)
```

```
Base class for all neural network modules.

    Your models should also subclass this class.

    Modules can also contain other Modules, allowing to nest them in
    a tree structure. You can assign the submodules as regular attribu
tes::

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

        class Model(nn.Module):
            def __init__(self):
                super(Model, self).__init__()
                self.conv1 = nn.Conv2d(1, 20, 5)
                self.conv2 = nn.Conv2d(20, 20, 5)

            def forward(self, x):
                x = F.relu(self.conv1(x))
                return F.relu(self.conv2(x))

    Submodules assigned in this way will be registered, and will have
their
    parameters converted too when you call :meth:`to`, etc.

    :ivar training: Boolean represents whether this module is in train
ing or
                    evaluation mode.
    :vartype training: bool
```

Часто, когда нейронная сеть не слишком сложная и последовательная, удобно пользоваться `nn.Sequential()` — последовательный контейнер из модулей. Модули, входящие в этот контейнер будут выполнятся последовательно один за другим, то есть выход одного подается на вход следующему.

Теперь напишем саму модель, которая

- Новый пункт
- Новый пункт

будет возвращать **логиты (logits)**. Логиты - это то, что получается **до применения SoftMax** для получения вероятностей в интервале 0-1. Они были у вас в логистической регрессии.

In [ ]:

```
# Создаем последовательную нейронную сеть
model = nn.Sequential()

# Преобразуем входной тензор размера batch_size x n1 x n2 x ... x nm
# к виду batch_size x n, где  n = n1 x n2 x ... x nm
model.add_module('flatten', nn.Flatten())

# Добавляем линейный слой с выходным размером 64.
# Размер входа равен произведению размерностей данных.
model.add_module('linear_1', nn.Linear(3 * 32 * 32, 64))

# Добавляем функцию активации ReLU
model.add_module('relu', nn.ReLU())

# Добавляем еще 1 линейный слой с выходным размером 10,
# равным количеству классов, на выходе получаем логиты
model.add_module('linear_2', nn.Linear(64, 10))
```

In [ ]:

```
# По-другому с именами слоев
model = nn.Sequential(OrderedDict([
    ('flatten', nn.Flatten()),
    ('linear_1', nn.Linear(3 * 32 * 32, 64)),
    ('relu', nn.ReLU()),
    ('linear_2', nn.Linear(64, 10))
]))
```

In [ ]:

```
# По-другому без имен слоев
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(3 * 32 * 32, 64),  # 3072 x 64
    nn.ReLU(),
    nn.Linear(64, 10)
)
```

In [ ]:

```
model.to(device)
```

Out[56]:

```
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=3072, out_features=64, bias=True)
  (2): ReLU()
  (3): Linear(in_features=64, out_features=10, bias=True)
)
```

Примечание: Если вы используете имена для слоев при создании контейнера, то имена должны быть разными, иначе при встрече слоя с уже существующим именем, предыдущий слой с таким именем будет перезаписан на новый.

У полученной модели можно посмотреть на все ее обучаемые параметры : `model.parameters()`

In [ ]:

```
print("Weight shapes:", [w.shape for w in model.parameters()])
```

```
Weight shapes: [torch.Size([64, 3072]), torch.Size([64]), torch.Size
([10, 64]), torch.Size([10])]
```

Мы создали модель, то есть научились по какому-либо входу получать выход модели. Чтобы обучить данную модель, нам нужно минимизировать эмпирический риск, то есть функцию потерь ("лосс"). Для этого определим его.

2.2 Объявление функции потерь (лосса)

Напишем функцию, которая будет принимать объекты из батча и их отклики и возвращать значение лосса. В качестве лосса будем использовать кросс-энтропию (log-loss или минус лог функция правдоподобия).

Если не пишите лоссы сами, а используете уже написанные из `torch.nn.functional (F)` , то внимательно читайте (найдите примеры), что они принимают на вход.

Например, `F.cross_entropy` и `nn.CrossEntropy` принимают на вход логиты (не вероятности!) и истинные метки классов (не one-hot вектора).

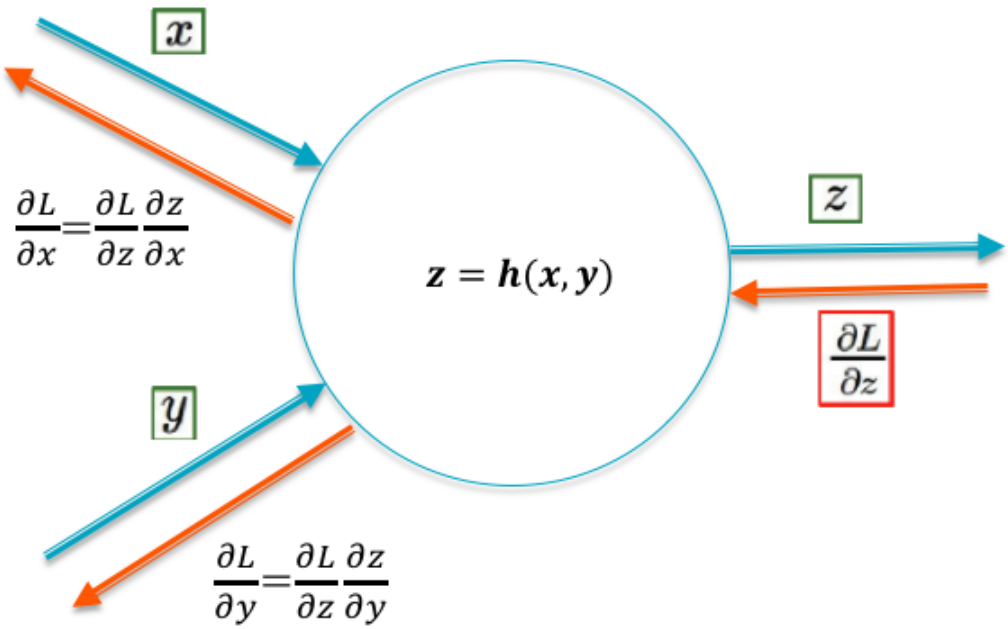
In [ ]:

```
# def compute_loss(logits, y_batch):
#     return F.cross_entropy(logits, y_batch).mean()

criterion = nn.CrossEntropyLoss()
```

Замечание: Многие лоссы, в том числе бинарную кросс-энтропию, можно найти и в `torch.nn.functional (F)` в виде функции, и в `torch.nn` в виде отдельного слоя.

2.3 Подсчет градиентов по всем обучаемым параметрам

Нейросети обучаются с помощью метода **Backpropagation**. Он объяснялся на лекции, а также с ним вы будете иметь дело в домашнем задании каждый раз, когда будете вычислять backward для каждого конкретного слоя.



Примечание: вообще говоря, сам метод обновления весов нейросети не обязан быть gradient-based, каким является backprop. Например, это могут быть эволюционные методы, или относительно недавний Equilibrium propagation, см. ответ на StackOverflow (https://stackoverflow.com/questions/55287004/are-there-alternatives-to-backpropagation).

2.4 PyTorch optimizers

В `torch.optim` лежит много разных уже готовых оптимизаторов таких как SGD, RMSprop, Adam и прочие.

Оптимизатор принимает набор тензоров, по которым он будет считать градиенты и которые будет оптимизировать. Обычно это все параметры модели поэтому обычно передаем `model.parameters()`. Сначала нам нужно сделать обратный проход посчитав все градиенты, потом мы делаем шаг и уже в конце зануляем градиенты.

In [ ]:

```
opt = torch.optim.SGD(model.parameters(), lr=0.01)

# loss.backward()      # обратный проход, считаем градиенты
# opt.step()           # делаем шаг градиентного спуска
# opt.zero_grad()      # зануляем градиенты
```

2.5 Цикл обучения нейросети: реализация

Теперь у нас все готово для обучения.

В фазе обучения мы вызываем метод `train(True)` у модели, чтобы перевести ее в фазу обучения: `model.train(True)`. В фазе тестирования ставим `model.train(False)` или `model.eval()`. Это влияет на:

- поведение Dropout слоев
- поведение BatchNorm слоев

Сейчас эти слои мы не используем, они будут на следующем занятии, но уже держим это в голове.

В фазе тестирования будем использовать контекстный менеджр `torch.no_grad` , который отключает возможность подсчета градиентов. Это позволяет более экономно использовать память.

По ходу обучения будем сохранять лучшую модель по метрике на валидации. Это просто делается с помощью метода `torch.save` . Рекомендуется сохранять не сам класс модели, а ее состояние (значения параметров). Подробности можно посмотреть здесь (https://pytorch.org/tutorials/beginner/saving_loading_models.html). Выгрузить модель можно с помощью метода `torch.load()` .

In [ ]:

```
num_epochs = 100  # общее кол-во полных проходов ("эпох") по обучаемым данным
batch_size = 64  # кол-во объектов в одном батче

num_train_batches = len(X_train) // batch_size
num_val_batches = len(X_val) // batch_size

history = defaultdict(lambda: defaultdict(list))

best_val_acc = 0.

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 batch_generator(X_train, y_train, batch_size, devic

        # Обучаемся на батче (одна "итерация" обучения нейросети)
        logits = model(X_batch)
        loss = criterion(logits, y_batch)
        # Обратный проход, шаг оптимизатора и зануление градиентов
        loss.backward()
        opt.step()
        opt.zero_grad()

        # Используйте методы тензоров:
        # detach -- для отключения подсчета градиентов
        # cpu -- для перехода на cpu
        # numpy -- чтобы получить numpy массив
        train_loss += loss.detach().cpu().numpy()
        y_pred_np = np.argmax(logits.detach().cpu().numpy(), axis=1)
        y_batch_np = y_batch.cpu().numpy()
        train_acc += (y_batch_np == y_pred_np).sum()

    # Подсчитываем лоссы и сохраням в "историю"
    train_loss /= num_train_batches
    train_acc /= num_train_batches * batch_size
    history['loss']['train'].append(train_loss)
    history['acc']['train'].append(train_acc)

    # Устанавливаем поведение dropout / batch_norm  в тестирование
    model.eval()

    # Полный проход по валидации
    with torch.no_grad(): # Отключаем подсчет градиентов, то есть detach не нуж
        for X_batch, y_batch in batch_generator(X_val, y_val, batch_size, devic
            logits = model(X_batch)
            loss = criterion(logits, y_batch)

            val_loss += loss.cpu().numpy().sum()
            y_pred_np = np.argmax(logits.cpu().numpy(), axis=1)
            y_batch_np = y_batch.cpu().numpy()
```

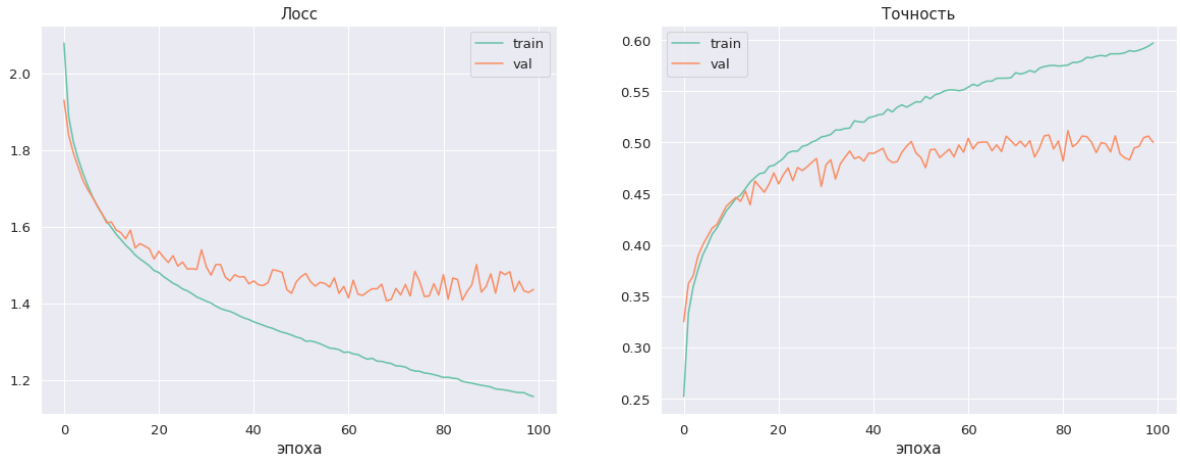
```
            val_acc += (y_batch_np == y_pred_np).sum()

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

    # Сохраняем лучшую модель по метрике на валидации
    if val_acc > best_val_acc:
        torch.save(model.state_dict(), 'first_model.pth')

    # 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))
```

```
Epoch 1 of 100 took 1.167s
  training loss (in-iteration):         2.079922
  validation loss (in-iteration):       1.930820
  training accuracy:                    25.21 %
  validation accuracy:                  32.51 %
Epoch 2 of 100 took 1.007s
  training loss (in-iteration):         1.887373
  validation loss (in-iteration):       1.839564
  training accuracy:                    33.38 %
  validation accuracy:                  36.24 %
Epoch 3 of 100 took 1.017s
  training loss (in-iteration):         1.821511
  validation loss (in-iteration):       1.792492
  training accuracy:                    35.92 %
  validation accuracy:                  37.00 %
Epoch 4 of 100 took 1.028s
  training loss (in-iteration):         1.777731
  validation loss (in-iteration):       1.756482
  training accuracy:                    37.56 %
```

Построим кривые обучения (learning curves):

In [ ]:

```
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.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.xlabel('эпоха', fontsize=15)
plt.legend();
```



Заметки

- Не забывайте **занулять градиенты** после каждой итерации.
- Если ваш loss стал `nan` / `inf`, то повыводите то, что происходит на каждой итерации и поймите в каком именно месте проблема.
- Если ваш loss уменьшался, а потом стал равен `nan`, то попробуйте уменьшить `learning rate`.

2.6 Модульный стиль создания нейросети

Сделаем аналогичную модель, только с 3 слоями, вместо 2, и в виде модуля, а не Sequential-модели. Заметим, что здесь наша модель - это модуль, который наследуется от `nn.Module` и содержит в себе другие модули, такие как `nn.Linear`.

Как было сказано выше, модуль должен обязательно иметь метод `forward()`, который мы сами определяем. Метод `backward()` является необязательным, PyTorch сможет сам понять, что делать при обратном проходе.

In [ ]:

```python
class MySimpleModel(nn.Module):
    def __init__(self):
        '''
        Здесь объявляем все слои, которые будем использовать
        '''
        super(MySimpleModel, self).__init__()
        self.linear1 = nn.Linear(3 * 32 * 32, 256)
        self.linear2 = nn.Linear(256, 64)
        self.linear3 = nn.Linear(64, 10)

    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))
        x = self.linear3(nn.ReLU()(x))
        return x
```

In [ ]:

```python
###### 1. Оберните цикл обучения нейросети в отдельную функцию ######

def train(
    model,
    criterion,
    optimizer,
    X_train, y_train,
    X_val, y_val,
    num_epochs=100,
    batch_size=64,
    model_path='model.pth'
):
    """
    # Обучение модели
    """

    num_train_batches = len(X_train) // batch_size
    num_val_batches = len(X_val) // batch_size

    history = defaultdict(lambda: defaultdict(list))

    best_val_acc = 0.

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

        start_time = time.time()

        model.train(True)  # устанавливаем поведение dropout / batch_norm  в об

        # На каждой "эпохе" делаем полный проход по данным
        for X_batch, y_batch in batch_generator(X_train, y_train, batch_size, d

            # Обучаемся на батче (одна "итерация" обучения нейросети)
            logits = model(X_batch)

            loss = criterion(logits, y_batch)

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

            train_loss += loss.detach().cpu().numpy()
            y_pred_np = np.argmax(logits.detach().cpu().numpy(), axis=1)
            y_batch_np = y_batch.cpu().numpy()
            train_acc += (y_batch_np == y_pred_np).sum()

        # Подсчитываем лоссы и сохраням в "историю"
        train_loss /= num_train_batches
        train_acc /= num_train_batches * batch_size
        history['loss']['train'].append(train_loss)
        history['acc']['train'].append(train_acc)

        # Устанавливаем поведение dropout / batch_norm  в тестирование
        model.eval()

```

```
        # Полный проход по валидации
        with torch.no_grad():
            for X_batch, y_batch in batch_generator(X_val, y_val, batch_size, d
                logits = model(X_batch)
                loss = criterion(logits, y_batch)

                val_loss += loss.cpu().numpy()
                y_pred_np = np.argmax(logits.cpu().numpy(), axis=1)
                y_batch_np = y_batch.cpu().numpy()
                val_acc += (y_batch_np == y_pred_np).sum()

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

        # Сохраняем лучшую модель по метрике на валидации
        if val_acc > best_val_acc:
            torch.save(model.state_dict(), model_path)

        # 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))

    return model, history
```

In [ ]:

```
###### 2. Оберните построение графиков в отдельную функцию    ######

def plot_learning_curves(history):
    """
    Построение графиков
    """
    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.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.xlabel('эпоха', fontsize=15)
    plt.legend()
```

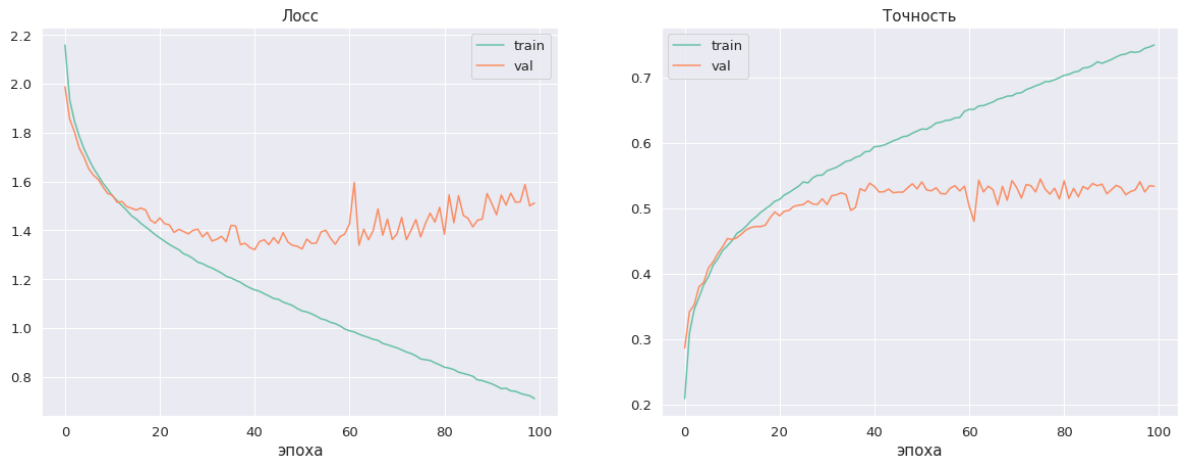
In [ ]:

```
###### 3. Обучите модель в Functional-стиле на CIFAR10     ######

model = MySimpleModel().to(device)

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

model, history = train(
    model, criterion, optimizer,
    X_train, y_train,
    X_val, y_val,
    num_epochs=100,
    batch_size=50,
    model_path='simple_model.pth'
)
```

```
Epoch 1 of 100 took 1.485s
  training loss (in-iteration):         2.158215
  validation loss (in-iteration):       1.987247
  training accuracy:                    20.94 %
  validation accuracy:                  28.66 %
Epoch 2 of 100 took 1.454s
  training loss (in-iteration):         1.934038
  validation loss (in-iteration):       1.855665
  training accuracy:                    30.90 %
  validation accuracy:                  34.19 %
Epoch 3 of 100 took 1.464s
  training loss (in-iteration):         1.846961
  validation loss (in-iteration):       1.802851
  training accuracy:                    34.52 %
  validation accuracy:                  35.25 %
Epoch 4 of 100 took 1.447s
  training loss (in-iteration):         1.785706
  validation loss (in-iteration):       1.736702
  training accuracy:                    36.34 %
```

In [ ]:

```
###### 4. Выведите графики функции потерь и метрики качества   ######

plot_learning_curves(history)
```



Что изменилось при увеличении числа слоев?