Phystech@DataScience

Домашнее задание 2

Правила, прочитайте внимательно:

- Выполненную работу нужно отправить телеграм-боту `@miptstats_pds_bot`. Для начала работы с ботом каждый раз отправляйте `/start`. **Работы, присланные иным способом, не принимаются.**
- Дедлайн см. в боте. После дедлайна работы не принимаются кроме случаев наличия уважительной причины.
- Прислать нужно ноутбук в формате `ipynb`.
- Выполнять задание необходимо полностью самостоятельно. **При обнаружении списывания все участники списывания будут сдавать устный зачет.**
- Решения, размещенные на каких-либо интернет-ресурсах, не принимаются. Кроме того, публикация решения в открытом доступе может быть приравнена к предоставлении возможности списать.
- Для выполнения задания используйте этот ноутбук в качестве основы, ничего не удаляя из него. Можно добавлять необходимое количество ячеек.
- Комментарии к решению пишите в markdown-ячейках.
- Выполнение задания (ход решения, выводы и пр.) должно быть осуществлено на русском языке.
- Если код будет не понятен проверяющему, оценка может быть снижена.
- Никакой код из данного задания при проверке запускаться не будет. *Если код студента не выполнен, недописан и т.д., то он не оценивается.*
- **Код из рассказанных на занятиях ноутбуков можно использовать без ограничений.**

Правила оформления теоретических задач:

- Решения необходимо прислать одним из следующих способов:
  - фотографией в правильной ориентации, где все четко видно, а почерк разборчив,
    - отправив ее как файл боту вместе с ноутбуком *или*
    - вставив ее в ноутбук посредством `Edit -> Insert Image` (фото, вставленные ссылкой, не принимаются);
  - в виде $\LaTeX$ в markdown-ячейках.
- Решения не проверяются, если какое-то требование не выполнено. Особенно внимательно все проверьте в случае выбора второго пункта (вставки фото в ноутбук). **Неправильно вставленные фотографии могут не передаться при отправке.** Для проверки попробуйте переместить `ipynb` в другую папку и открыть его там.
- В решениях поясняйте, чем вы пользуетесь, хотя бы кратко. Например, если пользуетесь независимостью, то достаточно подписи вида "*X и Y незав.*"
- Решение, в котором есть только ответ, и отсутствуют вычисления, оценивается в 0 баллов.

Баллы за задание:

- Задача 1 — 80 баллов
- Задача 2 — 30 баллов

Сверточные сети

В этой домашней работе вам предстоит построить сверточную сеть для классификации изображений.

Биология

Необходимо классифицировать изображения МРТ головного мозга из датасета **"Brain Tumor Classification (MRI)"** и определить вид опухоли.

Физика

Необходимо классифицировать изображения солнечного затмения из датасета **"Solar Eclipse Classification"** по степени: частичное, полное и кольцевое.

Задача 1.

Пожалуйста, ПРОЧИТАЙТЕ ВНИМАТЕЛЬНО то, что написано ниже, там изложены требования к вашей работе и полезные советы!

Требование к работе

- **Запрещено** использовать предобученные нейросети.

- **Запрещено** использовать тестовые данные где-либо за исключением вычисления финальной оценки качества. Подсказка — распределение данных на тесте такое же как в тестовых данных.

Советы

Архитектура нейросети

- В отличие от семинара в данном датасете могут встретиться картинки разных размеров. Эту проблему можно решить двумя способами:
  - Используя `torchvision.transforms.Resize` можно привести картинки к единому размеру. Если вы решите использовать этот способ, стоит посмотреть, какого в принципе размера встречаются картинки, чтобы не сжать их слишком сильно. Для картинок одного размера можно обучить бейзлайн в виде полносвязной сети.
  - Учесть переменный размер картинки в архитектуре сети. Общий принцип здесь такой: можно использовать свертки с нужным `padding`, чтобы не иметь проблем из-за уменьшения размеров картинки из-за сверток, последовательно применяя сверточные слои и пуллинги, нужно увеличивать количество каналов одновременно с уменьшением размера картинок (из-за пуллинга), а в конце, получив картинку размера (n_channels, nx, ny), оставить вектор размера (n_channels) (n_channels будет одинаковым для всех картинок, поскольку зависит от архитектуры сети!). Сделать это можно усреднением по пространственным картам torch.nn.AdaptiveAvgPool2d.
- Попробуйте разные размеры фильтров, страйдинг, паддинг
- Также можно попробовать разные активации: `tanh`, `leaky relu` и другие.

Процесс обучения

- Воспользуйтесь GPU google colab или любой другой GPU, которая у вас есть.
- Для сокращения вычислительной сложности можно поэксперементировать с параметром `stride`. Кроме того можете попробовать разные виды Poooling-ов.
- Помните, что некоторым нейросетям требуется 10 эпох, чтобы сойтись, а некоторым – 500. Большие нейросети дольше обучаются.
- Если вы достигли какого-то порога на валидации лучше подождать примерно 10 эпох перед тем как останавливать обучение.

И главное:

- Рисуйте кривые обучения: loss и метрика качества (лучше использовать F1-меру) для обучения и валидации.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import os
import shutil

import torch
from torch import nn

import torchvision
from torchvision import transforms

from sklearn.metrics import f1_score

from sklearn.utils.random import sample_without_replacement

from IPython.display import clear_output

from collections import defaultdict

from torch.optim import lr_scheduler

from matplotlib.animation import FuncAnimation, ImageMagickFileWriter
from IPython.display import Image, clear_output

import time

%matplotlib inline
```

Для Google Colab

Чтобы не грузить данные каждый раз в колаб при его отключении, а данные сюда грузятся небыстро, будет лучше всего поступить следующим образом.

- Загрузите архив на диск.
- Примонтируйте ваш диск к данному ноутбуку с помощью кода ниже

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

- В панели слева ("Файлы") откройте папку `drive/MyDrive/..` и найдите архив с файлом на диске
- Кликните по файлу и нажмите на кнопку "Скопировать путь"

Теперь вы можете обратиться к данным, используя скопированный путь

In [ ]:
```
# Путь до диска (для напоминания)
DISK_PATH = "drive/MyDrive"
# Путь до архива с данными (пример)
ZIP_PATH = "/content/drive/MyDrive/Ph@DS_ML/data.zip"
# Путь для папки с данными
DATA_PATH = "."
```

- Разархивируйте данные на диске.

In [ ]:
```
! unzip $ZIP_PATH -d  $DATA_PATH
```

In [ ]:
```
DATA_PATH = os.path.join(DATA_PATH, 'data')
```

В папке `DATA_PATH` теперь хранится папка с тренировочными данными (у биологов - `Training`, у физиков - `Train`) и тестовыми (у биологов - `Testing`, у физиков - `Test`). В папку с тестовыми данными не подглядывать :)

In [ ]:
```
! ls $DATA_PATH
```

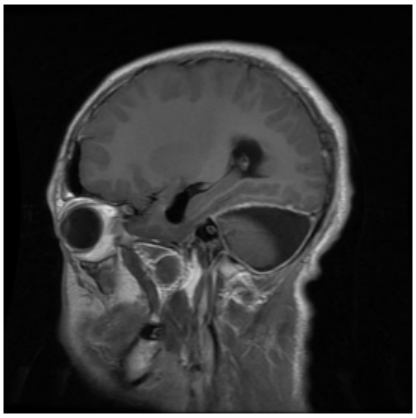
```
"ls" Ґ пў«пҐвбп ўгваҐҐ© Ё«Ё ўҐиҐ©
Є®¬ ¤®©, ЁбЇ®«пҐ¬®© Їа®Ја ¬¬®© Ё«Ё Ї ЄҐвл¬ д ©«®¬.
```

Подготовка датасетов

Биология

В `train` датасете 4 вида опухолей (или их отсутсвия) и 2870 изображений МРТ головного мозга (объектов). Посмотрим на какую-нибудь картину из набора данных.

In [ ]:
```
path_to_img = os.path.join(DATA_PATH, "training/glioma_tumor/gg (40).jpg")
image = plt.imread(path_to_img)
plt.figure(figsize=(12, 5))
plt.imshow(image)
plt.axis("off");
```



Разобьем `train` выборку на `train` и `val`:

In [ ]:
```
# Папка со всеми МРТ изображениями / папка с изображениями для тренировки
TRAIN_DIR = os.path.join(DATA_PATH, "training")
# Считываем названия директорий, которые и являются видом опухоли
TUMOR_LIST = {i:name for i, name in enumerate(os.listdir(TRAIN_DIR))}


# Папка с изображениями для валидации
VAL_DIR = os.path.join(DATA_PATH, "val")
os.makedirs(VAL_DIR, exist_ok=True)

# Папка с изображениями для теста
TEST_DIR = os.path.join(DATA_PATH, "testing")


# Доля изображений в валидации
VAL_FRAC = 0.3


# Создаем директорию с валидационной выборкой для каждого вида опухоли.
for tumor in TUMOR_LIST.values():
    os.makedirs(os.path.join(VAL_DIR, tumor), exist_ok=True)

    # Считываем выборку изображений.
    tumor_path = os.path.join(TRAIN_DIR, tumor)
```

```
    # Сортируем изображения для детерминированнсти
    images_filename = sorted(os.listdir(tumor_path))

    # Выделяем часть изображений для валидации
    # Выбираем случайные изображения из выборки для валидции, с установленным random_state
    num_images = len(images_filename)
    num_val = int(num_images * VAL_FRAC)
    indices = sample_without_replacement(num_images, num_val, random_state=42)
    val_images = np.take(images_filename, indices)

    print(f'{tumor} | train images = {num_images - num_val} | val images = {num_val}')

    # Сохраняем валидационную выборку
    for image_filename in val_images:
        source = os.path.join(TRAIN_DIR, tumor, image_filename)
        destination = os.path.join(VAL_DIR, tumor, image_filename)
        shutil.copy(source, destination)
        os.remove(source)
```

```
glioma_tumor | train images = 579 | val images = 247
meningioma_tumor | train images = 576 | val images = 246
no_tumor | train images = 277 | val images = 118
pituitary_tumor | train images = 579 | val images = 248
```

Данный датасет не очень сбалансирован по классам, возможные пути решения:

- *random oversampling* – включаем несколько копий объектов меньших классов, увеличивая их до размера большего класса, к копиям можно применять аугментацию;
- *random undersampling* – не включаем часть объектов больших классов в обучающую выборку;
- *примение весов* к лосс-функции;
- ...

Предлагаем вам сначала попробовать использовать обучить модель без решения проблемы несбалансированности (а вдруг и так все заработает?), а затем самим выбрать способ борьбы с ним и написать код в случае необходимости.

In [ ]:
```
#Делаем весовые коэффициенты к лосс-функции
counts = []

for tumor in TUMOR_LIST.values():
    # Считываем выборку изоражений
    tumor_path = os.path.join(TRAIN_DIR, tumor)

    images_filename = sorted(os.listdir(tumor_path))

    num_images = len(images_filename)
    counts = np.append(counts, num_images)

weights = sum(counts)/counts
```

Убедимся еще раз, что в папке train и val все разложено по папкам-классам (авторам). Эта структура папок важна для использования классов PyTorch по работе с данными (`ImageFolder` и `DataLoader`):

In [ ]:
```
!ls $TRAIN_DIR
```

In [ ]:
```
!ls $VAL_DIR
```

Физика

В `train` датасете 3 вида солнечного затмения и 2214 изображений Солнца (объектов). Посмотрим на какую-нибудь картину из набора данных.

In [ ]:
```
path_to_img = os.path.join(DATA_PATH, "train/Annular_solar_eclipse/Augmented_0__0_714.jpeg")
image = plt.imread(path_to_img)
plt.figure(figsize=(12, 5))
plt.imshow(image)
plt.axis("off");
```

Разобьем `train` выборку на `train` и `val`:

In [ ]:
```
# Папка со всеми фотографиями / папка с фотографиями для тренировки
TRAIN_DIR = os.path.join(DATA_PATH, "train")
# Считываем названия директорий, которые и являются видом затмения
ECLIPSE_LIST = {i:name for i, name in enumerate(os.listdir(TRAIN_DIR))}


# Папка с фотографиями для валидации
VAL_DIR = os.path.join(DATA_PATH, "val")
os.makedirs(VAL_DIR, exist_ok=True)

# Папка с фотографиями для теста
TEST_DIR = os.path.join(DATA_PATH, "test")


# Доля изображений в валидации
VAL_FRAC = 0.3


# Создаем директорию с валидационной выборкой для каждого вида затмения.
for eclipse in ECLIPSE_LIST.values():
    os.makedirs(os.path.join(VAL_DIR, eclipse), exist_ok=True)

    # Считываем выборку изображений.
    eclipse_path = os.path.join(TRAIN_DIR, eclipse)

    # Сортируем изображения для детерминированнсти
    images_filename = sorted(os.listdir(eclipse_path))

    # Выделяем часть изображений для валидации
    # Выбираем случайные изображения из выборки для валидции, с установленным random_state
    num_images = len(images_filename)
    num_val = int(num_images * VAL_FRAC)
    indices = sample_without_replacement(num_images, num_val, random_state=42)
    val_images = np.take(images_filename, indices)

    print(f'{eclipse} | train images = {num_images - num_val} | val images = {num_val}')

    # Сохраняем валидационную выборку
    for image_filename in val_images:
        source = os.path.join(TRAIN_DIR, eclipse, image_filename)
        destination = os.path.join(VAL_DIR, eclipse, image_filename)
        shutil.copy(source, destination)
        os.remove(source)
```

```
Annular_solar_eclipse | train images = 493 | val images = 211
Partial_solar_eclipse | train images = 543 | val images = 232
Total_solar_eclipse | train images = 515 | val images = 220
```

Убедимся еще раз, что в папке train и val все разложено по папкам-классам (авторам). Эта структура папок важна для использования классов PyTorch по работе с данными (`ImageFolder` и `DataLoader`):

In [ ]:
```
!ls $TRAIN_DIR
```

In [ ]:
```
!ls $VAL_DIR
```

In [ ]:
```
train_dataset = torchvision.datasets.ImageFolder(TRAIN_DIR, transform=transforms.Compose([transforms.ToTensor(),
                                                                                          transforms.Resize((128, 128))
val_dataset = torchvision.datasets.ImageFolder(VAL_DIR, transform=transforms.Compose([transforms.ToTensor(),
                                                                                      transforms.Resize((128, 128))])))
```

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

    :param history: (dict)
        f1 и loss на обучении и валидации
```

```
    '''
    # sns.set_style(style='whitegrid')
    fig = plt.figure(figsize=(20, 7))

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

    plt.subplot(1,2,2)
    plt.title('F1', fontsize=15)
    plt.plot(history['f1']['train'], label='train')
    plt.plot(history['f1']['val'], label='val')
    plt.ylabel('F1', fontsize=15)
    plt.xlabel('Эпоха', fontsize=15)
    plt.legend()
    plt.show()
```

In [ ]:
```
def plot_learning_curves_compar(histories, labels, y_max = None):
    '''
    Функция для вывода лосса и метрики во время обучения для нескольких моделей.

    :param histories: (dict)
        f1 и loss на обучении и валидации
    '''
    # sns.set_style(style='whitegrid')
    fig = plt.figure(figsize=(20, 7))
    i=0
    colors = ['blue', 'green', 'red', 'purple']
    for history in histories:
        plt.subplot(1,2,1)
        plt.title('Loss', fontsize=15)
        plt.plot(history['loss']['train'], label=labels[i]+', train', color=colors[i])
        plt.plot(history['loss']['val'], label=labels[i]+', val', color=colors[i], alpha=0.5)
        plt.ylabel('Loss', fontsize=15)
        plt.xlabel('Эпоха', fontsize=15)
        if y_max!=None:
            bottom, top = plt.ylim()
            plt.ylim(bottom, y_max)
        plt.legend()

        plt.subplot(1,2,2)
        plt.title('F1', fontsize=15)
        plt.plot(history['f1']['train'], label=labels[i]+', train', color=colors[i])
        plt.plot(history['f1']['val'], label=labels[i]+', val', color=colors[i], alpha=0.5)
        plt.ylabel('F1', fontsize=15)
        plt.xlabel('Эпоха', fontsize=15)

        i+=1

    plt.legend()
    plt.show()
```

In [ ]:
```
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) F1_score и loss на обучении и валидации ("история" обучения)
    '''
    all_time = 0
    history = defaultdict(lambda: defaultdict(list))

    for epoch in range(num_epochs):
        train_loss = 0
        train_f1 = 0
        train_for_f1_b = []
        train_for_f1_p = []
        val_loss = 0
        val_f1 = 0
```

```
        val_for_f1_b = []
        val_for_f1_p = []

        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 = transform_train(X_batch)

            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_for_f1_b = np.append(train_for_f1_b, y_batch.cpu().numpy())
            train_for_f1_p = np.append(train_for_f1_p, y_pred)

        # Подсчитываем лоссы и сохраням в "историю"
        train_loss /= len(train_batch_gen)
        train_f1 = f1_score(train_for_f1_b, train_for_f1_p, average="macro")
        history['loss']['train'].append(train_loss)
        history['f1']['train'].append(train_f1)

        # 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_for_f1_b = np.append(val_for_f1_b, y_batch.cpu().numpy())
            val_for_f1_p = np.append(val_for_f1_p, y_pred)

        # Подсчитываем лоссы и сохраням в "историю"
        val_loss /= len(val_batch_gen)
        val_f1 = f1_score(val_for_f1_b, val_for_f1_p, average="macro")
        history['loss']['val'].append(val_loss)
        history['f1']['val'].append(val_f1)

        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 f1: \t\t\t{:.2f} %".format(train_f1 * 100))
        print("  validation f1: \t\t\t{:.2f} %".format(val_f1 * 100))

        plot_learning_curves(history)

        all_time += (time.time() - start_time)

    return model, history, all_time
```

In [ ]:

```
batch_size = 256
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=batch_size, shuffle=True)
```

In [ ]:
```
device = 'cuda:0'
```

I. Построение сверточных сетей с использованием Dropout и BatchNorm

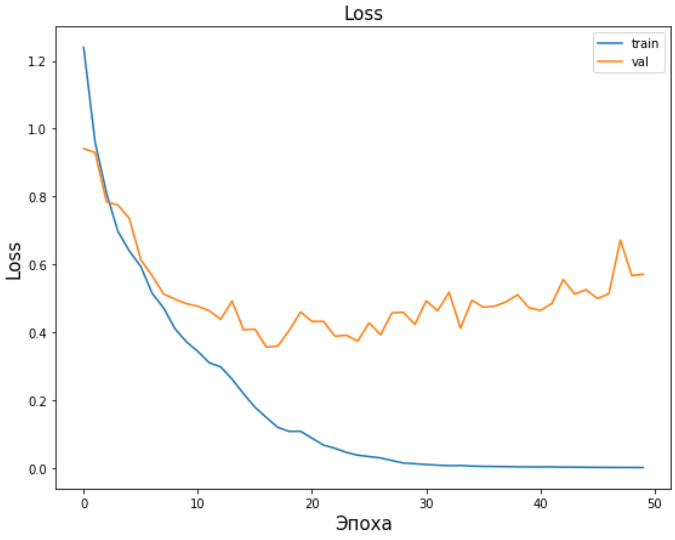
В первой части задания вам предстоит сравнить четыре различные реализации одной и той же сверточной сети. Для реализации сети можете смело использовать код с семинара.

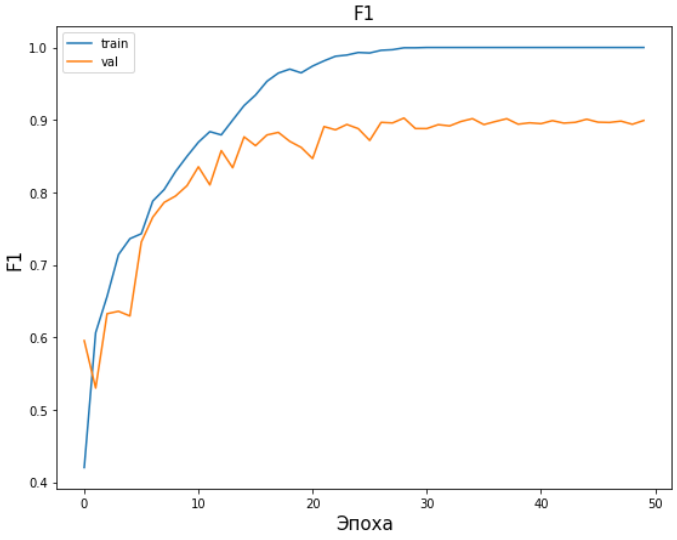
1. Постройте простую сверточную сеть без использования функций Dropout и BatchNorm.

In [ ]:
```
class SimpleConvNet_1(nn.Module):
    def __init__(self):
        super(SimpleConvNet_1, self).__init__()

        self.conv1 = nn.Conv2d(3, 32, 4)
        self.mp1 = nn.MaxPool2d(3)
        self.relu1 = nn.ReLU()

        self.conv2 = nn.Conv2d(32, 64, 4)
        self.mp2 = nn.MaxPool2d(3)
        self.relu2 = nn.ReLU()

        self.flatten = nn.Flatten()
        self.fc3 = nn.Linear(9216, 512)
        self.relu3 = nn.ReLU()
        self.fc4 = nn.Linear(512, len(weights))

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

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

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

In [ ]:
```
model = SimpleConvNet_1().to(device)
criterion = nn.CrossEntropyLoss(weight=torch.Tensor(weights), reduction='mean').to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

model, history, all_time = train(
    model, criterion, optimizer,
    train_loader, val_loader,
    num_epochs=50
)
print(all_time)
```

```
Epoch 50 of 50 took 20.148s
  training loss (in-iteration):         0.001521
  validation loss (in-iteration):       0.570988
  training f1:                  100.00 %
  validation f1:                        89.92 %
```






```
1069.3201513290405
```

In [ ]:
```
history_model_1 = history.copy()
```

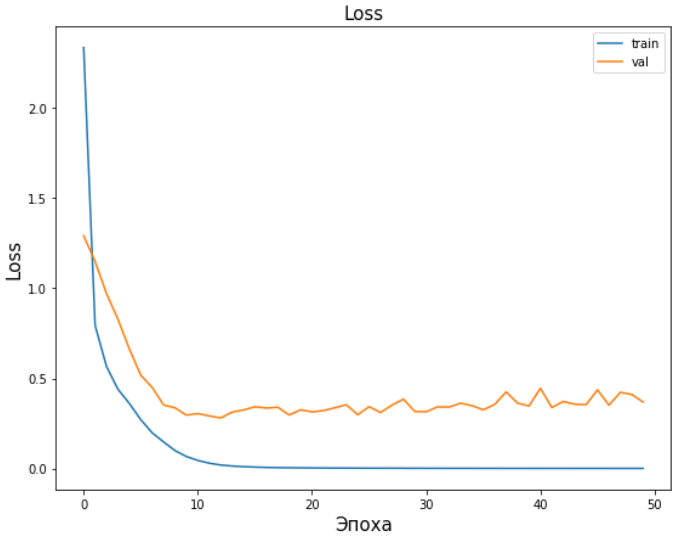
2. Попробуйте добавить BatchNorm на слои своей сверточной сети, не используя Dropout.

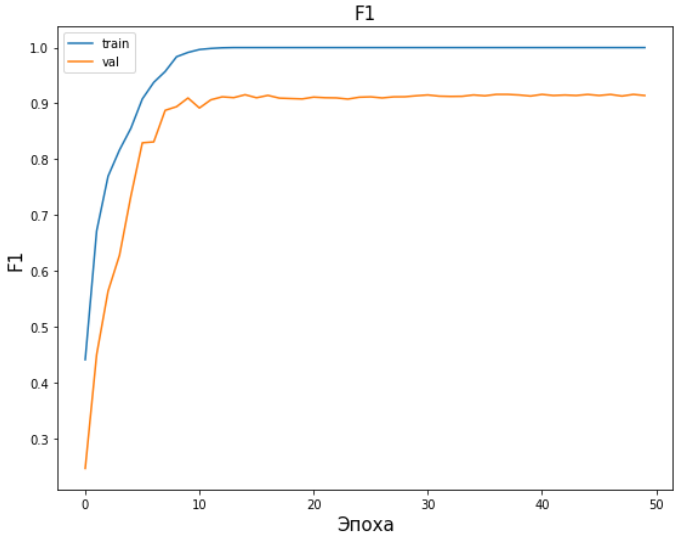
In [ ]:
```
class SimpleConvNet_2(nn.Module):
    def __init__(self):
```

```
        super(SimpleConvNet_2, self).__init__()

        self.conv1 = nn.Conv2d(3, 32, 4)
        self.mp1 = nn.MaxPool2d(3)
        self.bn1 = nn.BatchNorm2d(32)
        self.relu1 = nn.ReLU()

        self.conv2 = nn.Conv2d(32, 64, 4)
        self.mp2 = nn.MaxPool2d(3)
        self.bn2 = nn.BatchNorm2d(64)
        self.relu2 = nn.ReLU()

        self.flatten = nn.Flatten()
        self.fc3 = nn.Linear(9216, 512)
        self.relu3 = nn.ReLU()
        self.fc4 = nn.Linear(512, len(weights))

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

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

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

In [ ]:
```
model = SimpleConvNet_2().to(device)
criterion = nn.CrossEntropyLoss(weight=torch.Tensor(weights), reduction='mean').to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

model, history, all_time = train(
    model, criterion, optimizer,
    train_loader, val_loader,
    num_epochs=50
)
print(all_time)
```

```
Epoch 50 of 50 took 30.905s
  training loss (in-iteration):         0.000355
  validation loss (in-iteration):       0.368343
  training f1:                  100.00 %
  validation f1:                       91.41 %
```






```
1771.42032456398
```

In [ ]:
```
history_model_2 = history.copy()
```

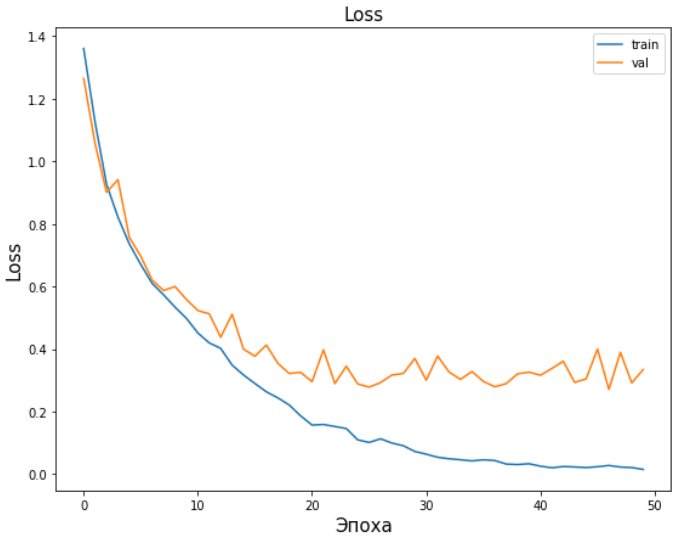
3. Попробуйте добавить Dropout на слои своей сверточной сети, не используя BatchNorm.

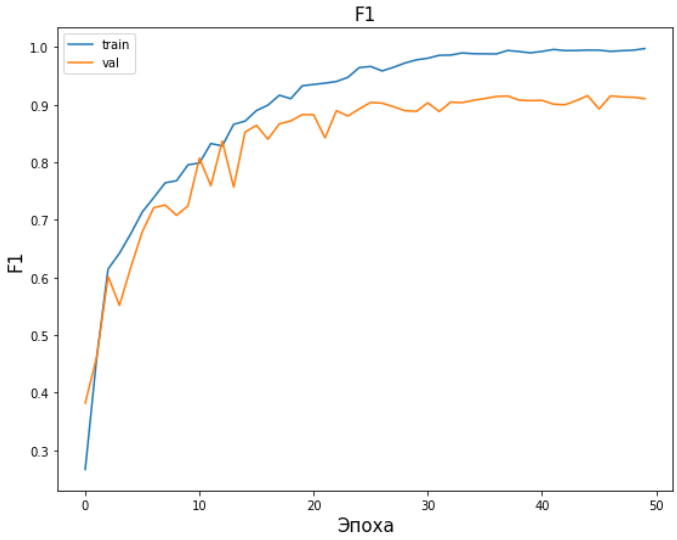
In [ ]:
```
class SimpleConvNet_3(nn.Module):
    def __init__(self):
        super(SimpleConvNet_3, self).__init__()

        self.conv1 = nn.Conv2d(3, 32, 4)
        self.mp1 = nn.MaxPool2d(3)
        self.droupout1 = nn.Dropout(0.2)
        self.relu1 = nn.ReLU()

        self.conv2 = nn.Conv2d(32, 64, 4)
        self.mp2 = nn.MaxPool2d(3)
        self.bn2 = nn.BatchNorm2d(64)
        self.droupout2 = nn.Dropout(0.2)
        self.relu2 = nn.ReLU()
```

```
        self.flatten = nn.Flatten()
        self.fc3 = nn.Linear(9216, 512)
        self.droupout3 = nn.Dropout(0.2)
        self.relu3 = nn.ReLU()
        self.fc4 = nn.Linear(512, len(weights))

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

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

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

In [ ]:
```
model = SimpleConvNet_3().to(device)
criterion = nn.CrossEntropyLoss(weight=torch.Tensor(weights), reduction='mean').to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

model, history, all_time = train(
    model, criterion, optimizer,
    train_loader, val_loader,
    num_epochs=50
)
print(all_time)
```

```
Epoch 50 of 50 took 21.448s
  training loss (in-iteration):         0.015260
  validation loss (in-iteration):       0.334519
  training f1:                  99.72 %
  validation f1:                        91.01 %
```






```
1080.3946461677551
```

In [ ]:
```
history_model_3 = history.copy()
```

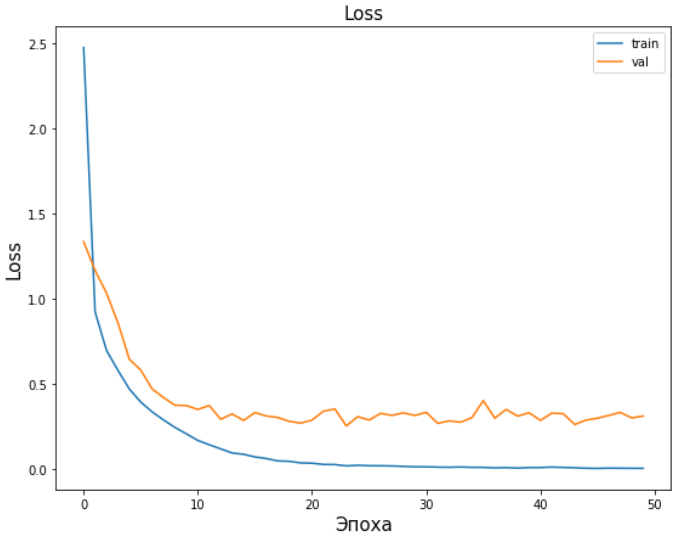
4. Теперь добавьте на все слои сети и Dropout, и BatchNorm.

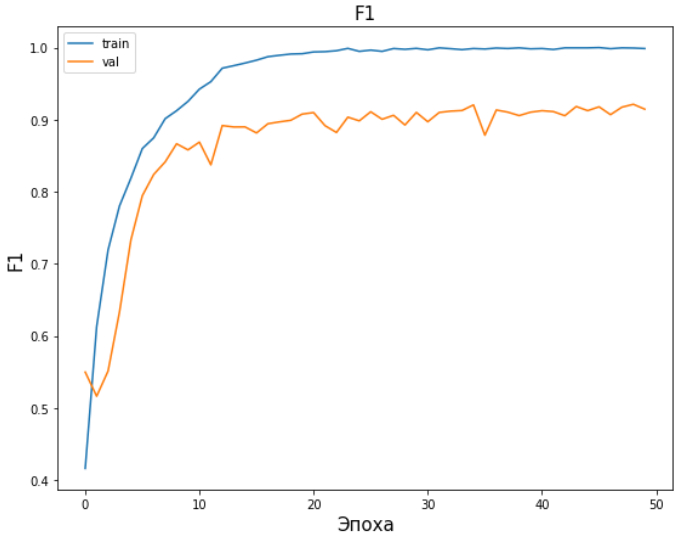
In [ ]:
```
class SimpleConvNet_4(nn.Module):
    def __init__(self):
        super(SimpleConvNet_4, self).__init__()

        self.conv1 = nn.Conv2d(3, 32, 4)
        self.mp1 = nn.MaxPool2d(3)
        self.bn1 = nn.BatchNorm2d(32)
        self.droupout1 = nn.Dropout(0.2)
        self.relu1 = nn.ReLU()

        self.conv2 = nn.Conv2d(32, 64, 4)
        self.mp2 = nn.MaxPool2d(3)
        self.bn2 = nn.BatchNorm2d(64)
        self.droupout2 = nn.Dropout(0.2)
        self.relu2 = nn.ReLU()

        self.flatten = nn.Flatten()
        self.fc3 = nn.Linear(9216, 512)
        self.droupout3 = nn.Dropout(0.2)
        self.relu3 = nn.ReLU()
        self.fc4 = nn.Linear(512, len(weights))

    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 [ ]:
```
model = SimpleConvNet_4().to(device)
criterion = nn.CrossEntropyLoss(weight=torch.Tensor(weights), reduction='mean').to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

model, history, all_time = train(
    model, criterion, optimizer,
    train_loader, val_loader,
    num_epochs=50
)
print(all_time)
```

```
Epoch 50 of 50 took 30.506s
  training loss (in-iteration):         0.005010
  validation loss (in-iteration):       0.311900
  training f1:                  99.87 %
  validation f1:                        91.45 %
```





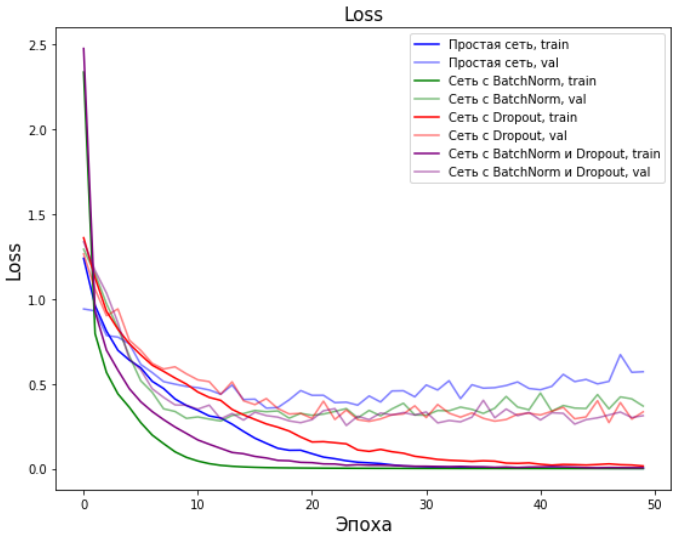

```
2341.410665988922
```

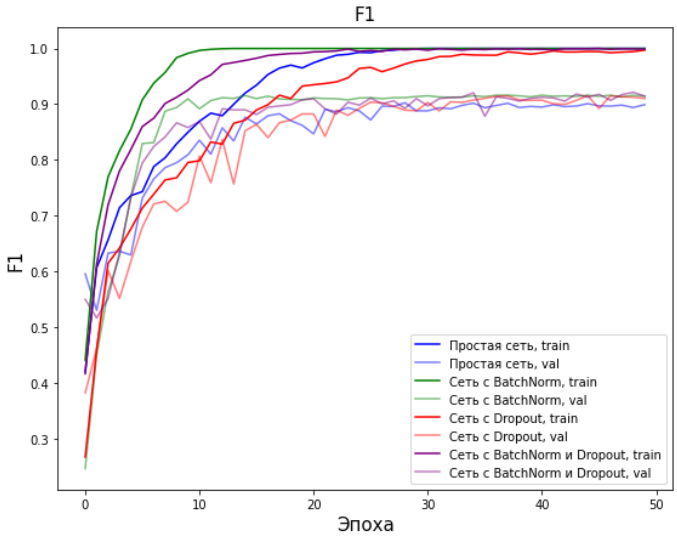
In [ ]:
```
history_model_4 = history.copy()
```

Проведите сравнение построенных сверточных сетей.

Для наглядности сравнения после обучения всех четырех сетей постройте общий график лосса и метрики качества на обучающей и валидационной выборках.

In [ ]:
```
plot_learning_curves_compar([history_model_1, history_model_2, history_model_3, history_model_4],
                            ['Простая сеть', 'Сеть с BatchNorm', 'Сеть с Dropout', 'Сеть с BatchNorm и Dropout'])
```





Вывод

Делаем вывод, что у нас сети очень быстро выходят на 100% обучение на трейне. И начинают переобучаться. Особенно это видно на обычной чети без Dropout'a и BatchNorm. Добавление их на слои сети помогает избежать настолько быстрого роста лосса на

валидации. При этом сеть только с Dropout'ом вышла на плато быстрее всего.

II. Сравнение различных оптимизаторов

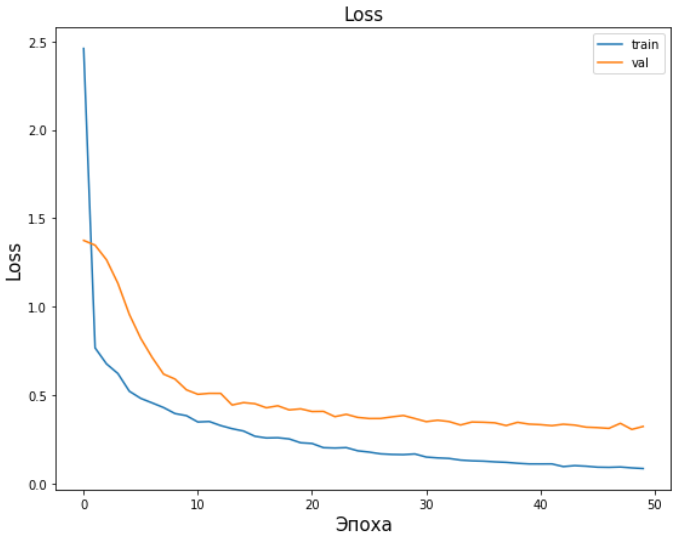
Обучите несколько нейронных сетей с различными оптимизаторами. Например, можно использовать `SGD`, `rmsprop`, `adam`, `adagrad` из `torch.optim`.

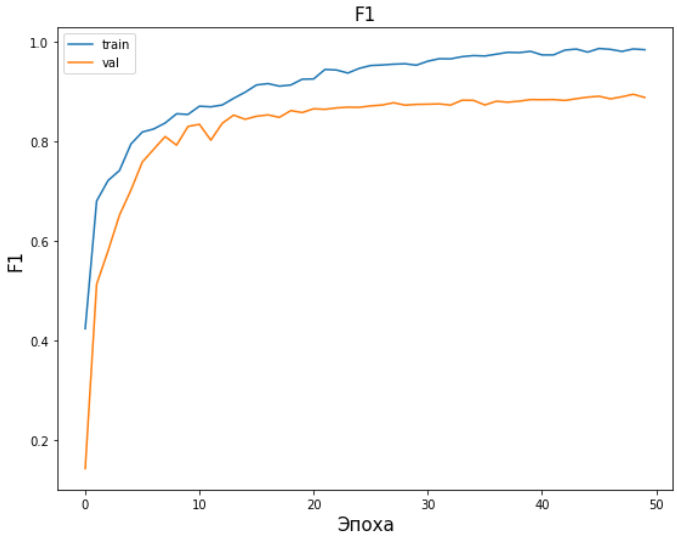
Проведите сравнение работы данных нейронных сетей и выберете лучший оптимизатор для вашей конкретной задачи.

In [ ]:
```
# Adam уже посмотрели в прошлом пункте и будем использовать history_model_4 для сравнения
```

In [ ]:
```
model = SimpleConvNet_4().to(device)
criterion = nn.CrossEntropyLoss(weight=torch.Tensor(weights), reduction='mean').to(device)
optimizer = torch.optim.Adagrad(model.parameters(), lr=0.001)

model, history, all_time = train(
    model, criterion, optimizer,
    train_loader, val_loader,
    num_epochs=50
)
print(all_time)
```

```
Epoch 50 of 50 took 23.046s
  training loss (in-iteration):         0.084298
  validation loss (in-iteration):       0.322283
  training f1:                  98.43 %
  validation f1:                        88.84 %
```



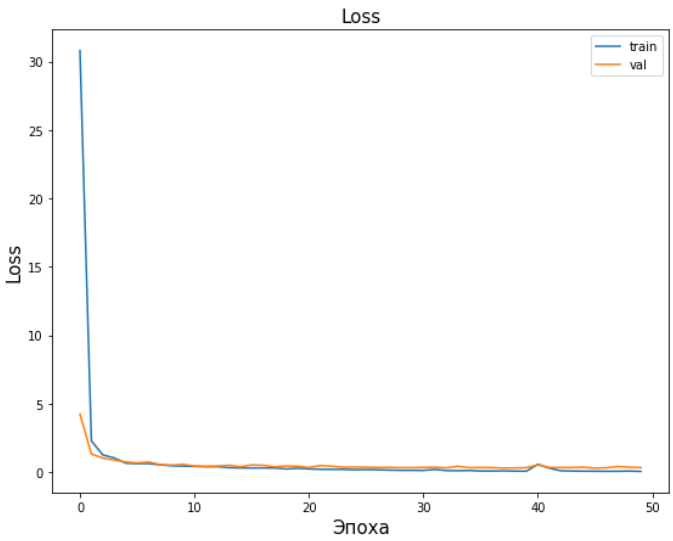


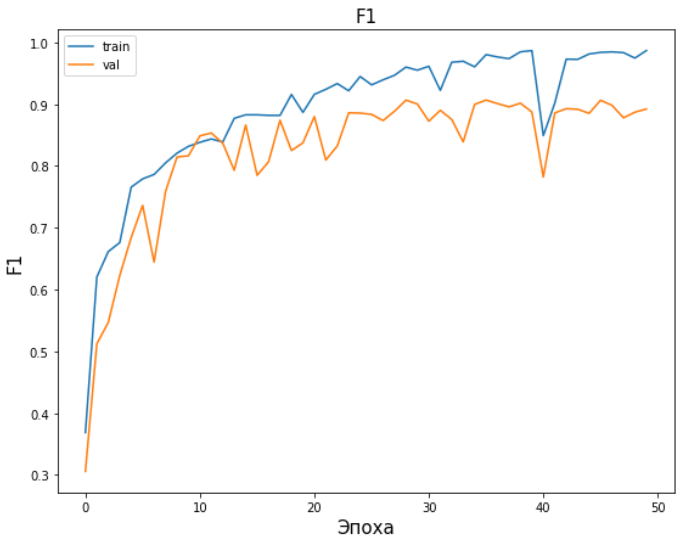

```
1146.5103738307953
```

In [ ]:
```
history_model_adagrad = history.copy()
```

In [ ]:
```
model = SimpleConvNet_4().to(device)
criterion = nn.CrossEntropyLoss(weight=torch.Tensor(weights), reduction='mean').to(device)
optimizer = torch.optim.RMSprop(model.parameters(), lr=0.001)

model, history, all_time = train(
    model, criterion, optimizer,
    train_loader, val_loader,
    num_epochs=50
)
print(all_time)
```

```
Epoch 50 of 50 took 20.541s
  training loss (in-iteration):         0.049644
  validation loss (in-iteration):       0.327572
  training f1:                  98.68 %
  validation f1:                        89.22 %
```



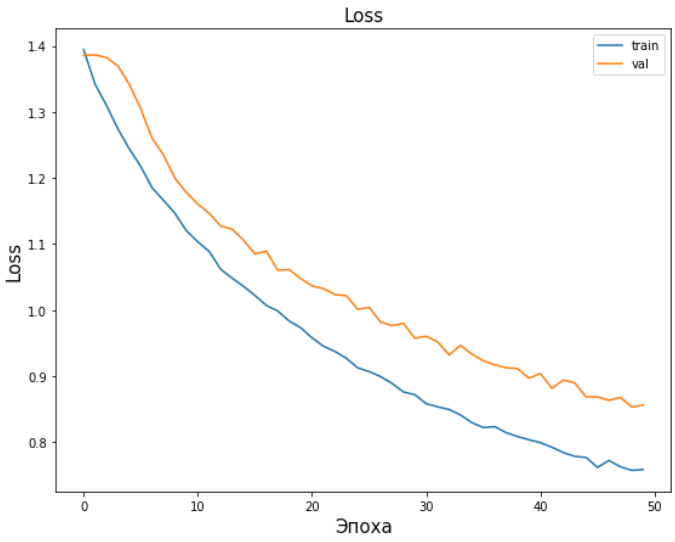


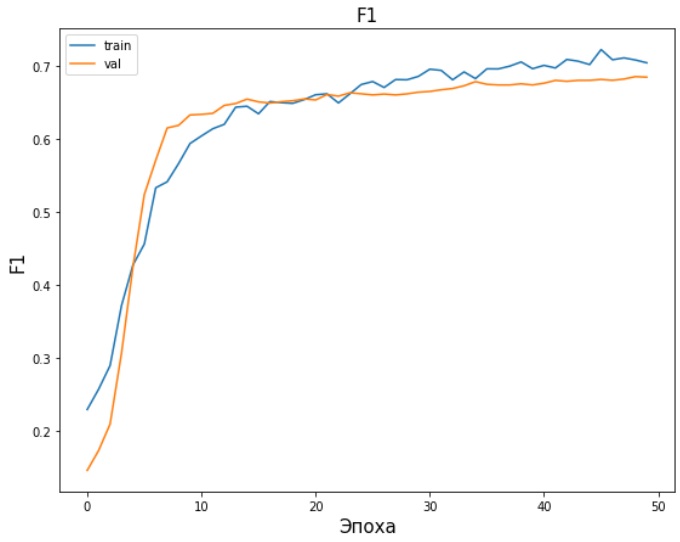

```
1123.3008217811584
```

In [ ]:
```
history_model_rms = history.copy()
```

In [ ]:
```
model = SimpleConvNet_4().to(device)
criterion = nn.CrossEntropyLoss(weight=torch.Tensor(weights), reduction='mean').to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)

model, history, all_time = train(
    model, criterion, optimizer,
    train_loader, val_loader,
    num_epochs=50
)
print(all_time)
```

```
Epoch 50 of 50 took 22.639s
  training loss (in-iteration):         0.758778
  validation loss (in-iteration):       0.856267
  training f1:                  70.37 %
  validation f1:                        68.41 %
```





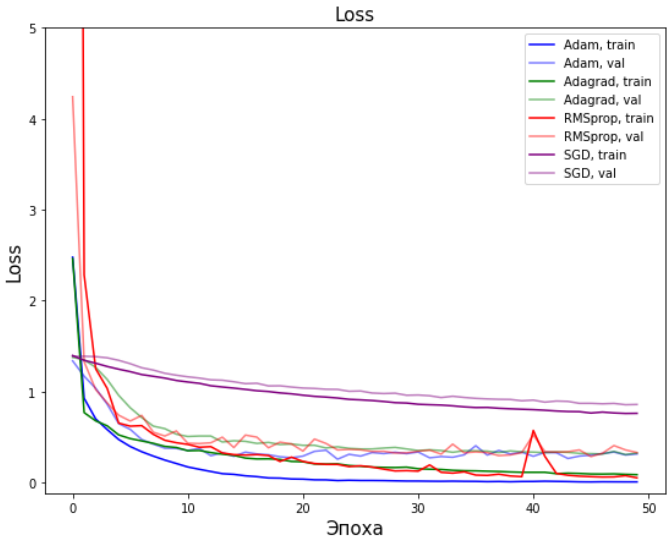

```
1133.0661790370941
```

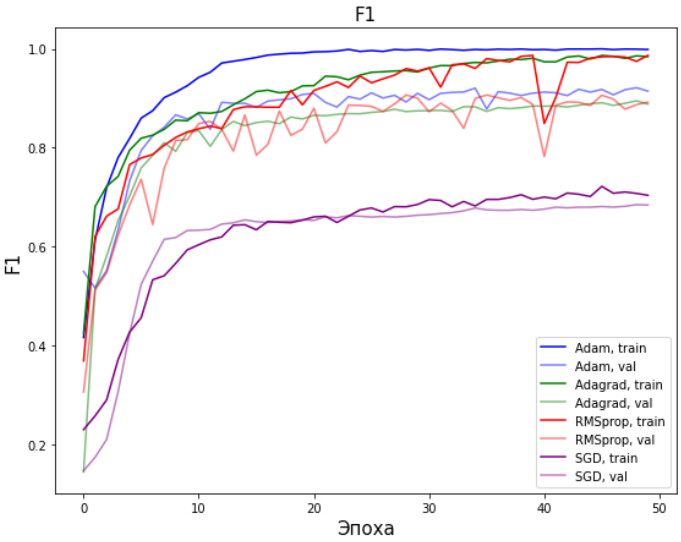
In [ ]:
```
history_model_sgd = history.copy()
```

Проведите сравнение построенных сверточных сетей.

Для наглядности сравнения после обучения всех сетей с различными оптимизаторами постройте общий график лосса и метрики качества на обучающей и валидационной выборках.

In [ ]:
```
plot_learning_curves_compar([history_model_4, history_model_adagrad, history_model_rms, history_model_sgd],
                            ['Adam', 'Adagrad', 'RMSprop', 'SGD'], 5)
```





Вывод

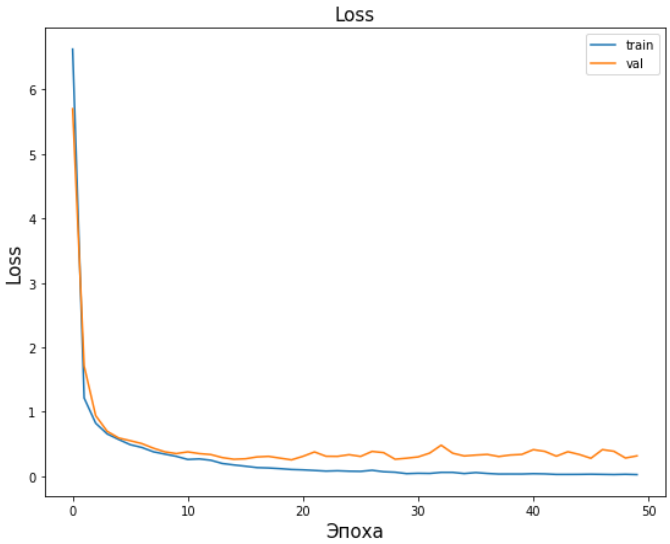
Делаем вывод, то SGD очень медленно сходится на таком фиксированном learning_rate. RMSprop очень нестабилен и сильно скачет (особенно по значению метрики качества). Adam лучше всего себя показал: самый стабильный и быстрый.

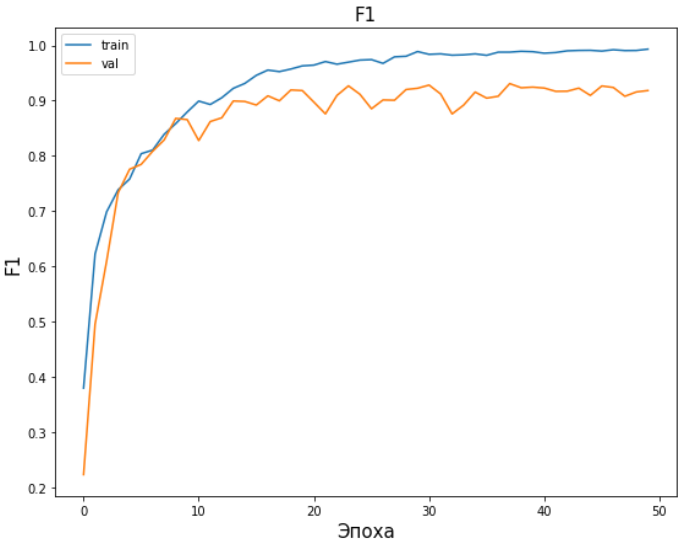
Для лучшего оптимизатора попробуйте поварьировать значение параметров (например, `learning rate`, `momentum` и так далее). Четырех различных значений для каждого параметра будет достаточно. Сравните результаты обучения, построив аналогичные общие графики лосса и метрики качества.

In [ ]:
```
# Adam с learnong_rate == 0.01 уже посмотрели в прошлом пункте и будем использовать history_model_4 для сравнения
```

In [ ]:
```
model = SimpleConvNet_4().to(device)
criterion = nn.CrossEntropyLoss(weight=torch.Tensor(weights), reduction='mean').to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.005)

model, history, all_time = train(
    model, criterion, optimizer,
    train_loader, val_loader,
    num_epochs=50
)
print(all_time)
```

```
Epoch 50 of 50 took 31.230s
  training loss (in-iteration):         0.025618
  validation loss (in-iteration):       0.316317
  training f1:                  99.32 %
  validation f1:                        91.83 %
```



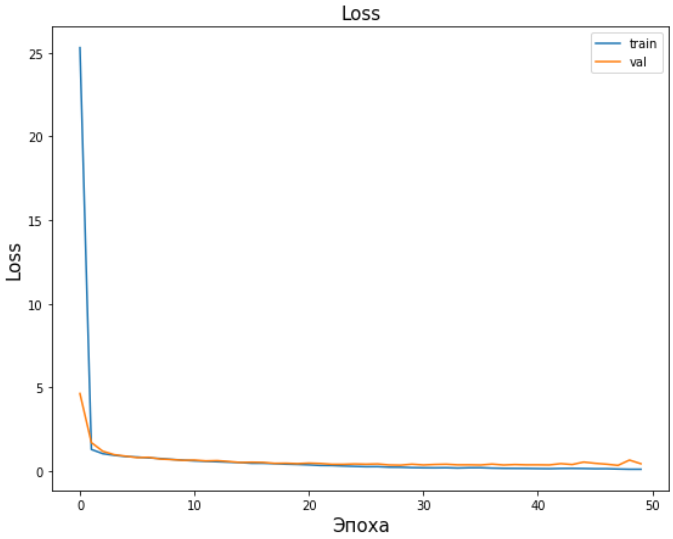


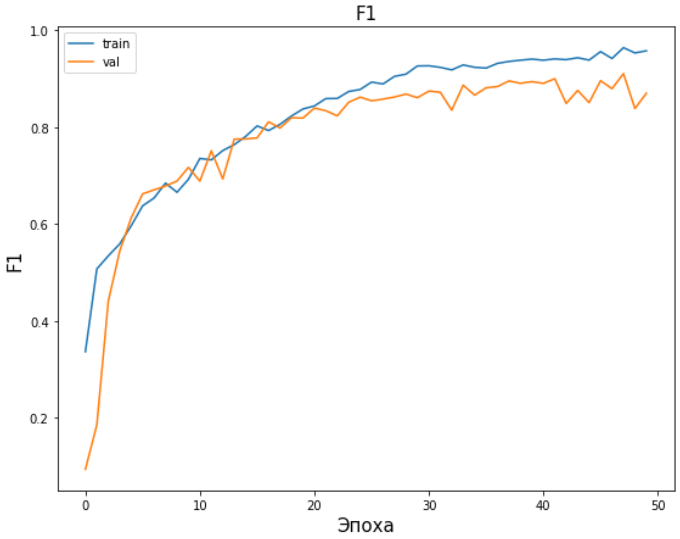

```
1225.054129600525
```

In [ ]:
```
history_model_005 = history.copy()
```

In [ ]:
```
model = SimpleConvNet_4().to(device)
criterion = nn.CrossEntropyLoss(weight=torch.Tensor(weights), reduction='mean').to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

model, history, all_time = train(
    model, criterion, optimizer,
    train_loader, val_loader,
    num_epochs=50
)
print(all_time)
```

```
Epoch 50 of 50 took 22.614s
  training loss (in-iteration):         0.099446
  validation loss (in-iteration):       0.428174
  training f1:                  95.70 %
  validation f1:                        86.94 %
```



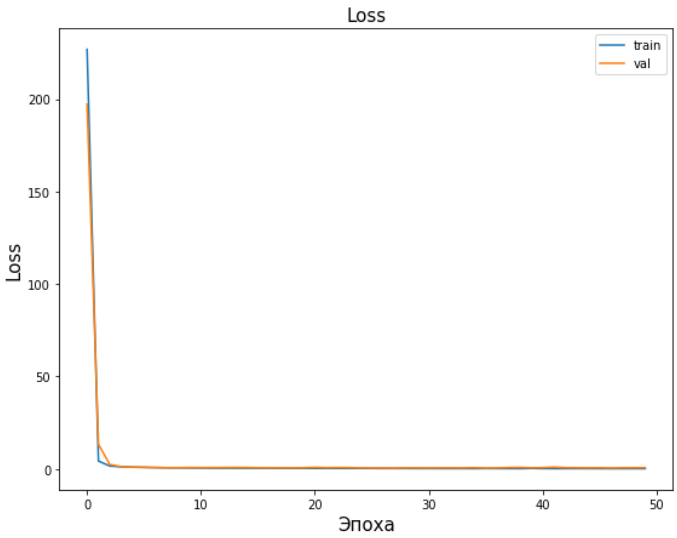


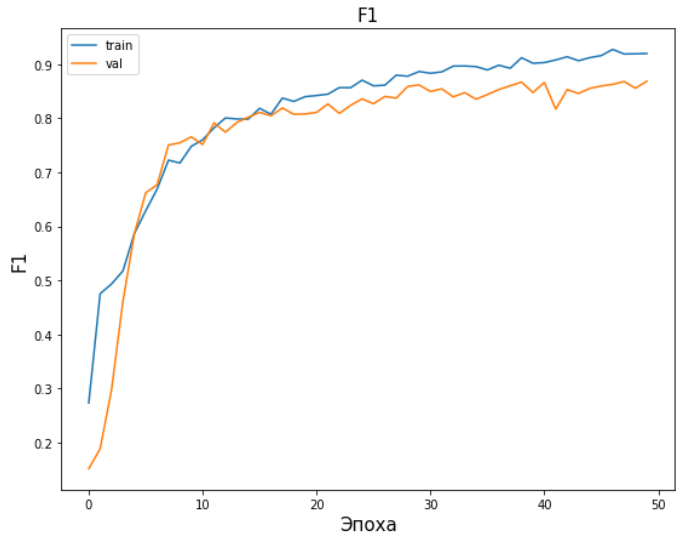

```
1151.4070620536804
```

In [ ]:
```
history_model_01 = history.copy()
```

In [ ]:
```
model = SimpleConvNet_4().to(device)
criterion = nn.CrossEntropyLoss(weight=torch.Tensor(weights), reduction='mean').to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.05)

model, history, all_time = train(
    model, criterion, optimizer,
    train_loader, val_loader,
    num_epochs=50
)
print(all_time)
```

```
Epoch 50 of 50 took 18.850s
  training loss (in-iteration):         0.181544
  validation loss (in-iteration):       0.681813
  training f1:                  91.97 %
  validation f1:                        86.83 %
```

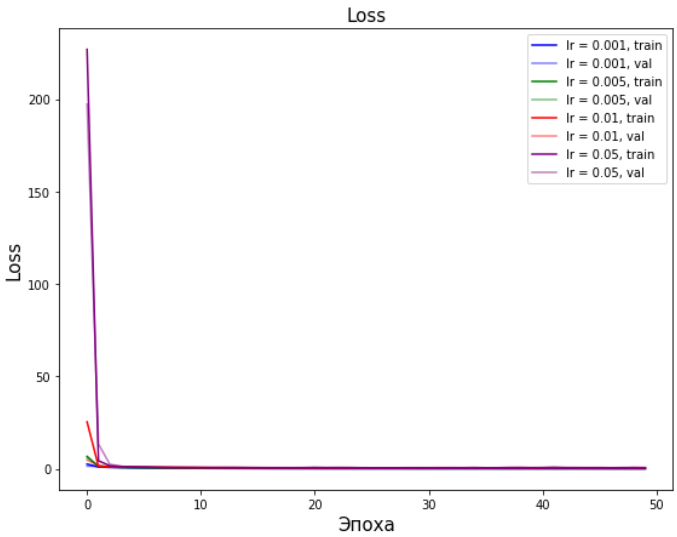


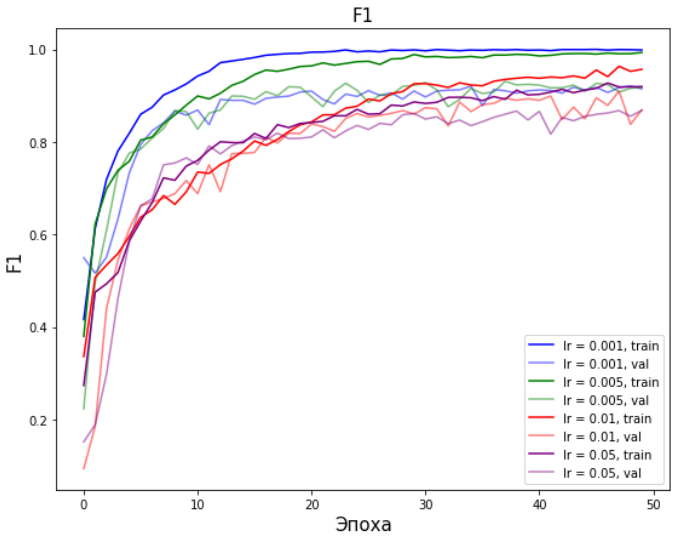



```
1157.5591251850128
```

In [ ]:
```
history_model_05 = history.copy()
```

In [ ]:
```
plot_learning_curves_compar([history_model_4, history_model_005, history_model_01, history_model_05],
                            ['lr = 0.001', 'lr = 0.005', 'lr = 0.01', 'lr = 0.05'])
```





Вывод

Сравниваем графики и видим, чтоо стабильнее всего без особых скачков и быстро сходится модель с низкой скоростью обучения. Мы более плавно идем к минимуму, а не скачем вокруг него, как происходит при более значительной скорости обучения.

III. Способы борьбы с переобучением

Мы знаем, что нейронные сети часто в какой-то момент начинают переобучаться. Есть целый набор различных способов борьбы с данным явлением.

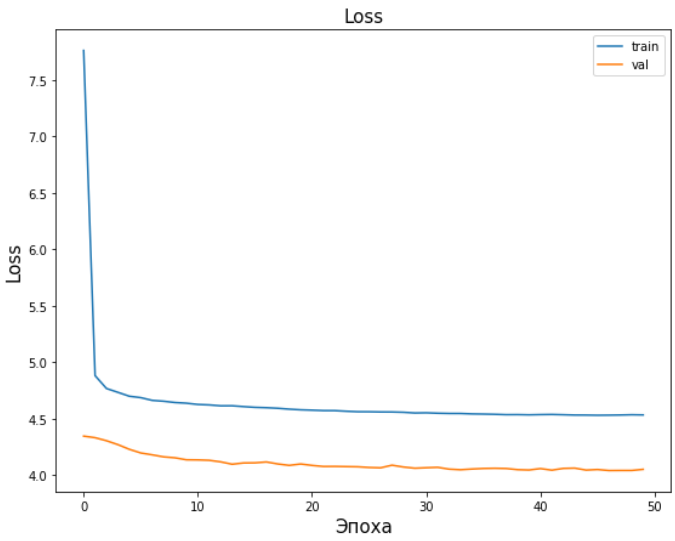
В данном пункте вам предлагается попробовать использовать две различные методики: `torch.optim.lr_scheduler` и `Label Smoothing`.

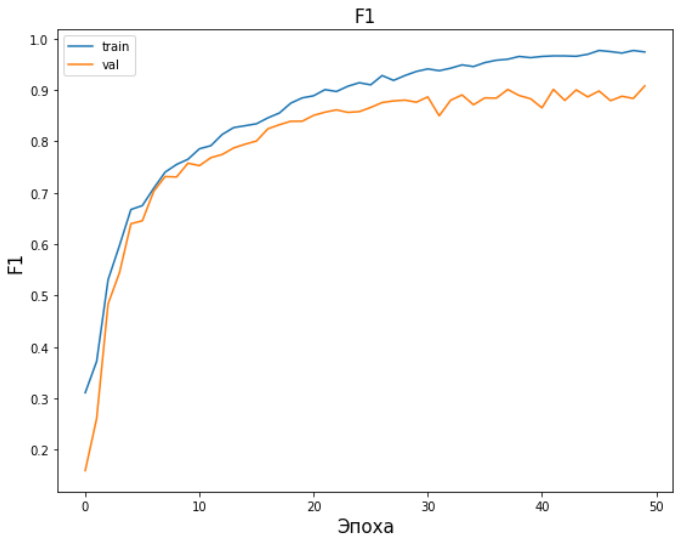
Обучите еще две сверточные сети, пользуясь данными методами борьбы с переобучением, и сравните их результаты со своей лучшей обученной нейронной сетью из прошлого пункта.

In [ ]:
```
# Добавляем Label Smoothing
class LabelSmoothingCrossEntropyLoss(nn.Module):
    def __init__(self, num_classes, weight, reduction, epsilon=0.01):
        super().__init__()
        self.num_classes = num_classes
        self.epsilon = epsilon
        self.criterion = nn.CrossEntropyLoss(weight, reduction)
        self.log_softmax = nn.LogSoftmax(dim=1)

    def forward(self, preds, target):
        cross_entropy_loss = self.criterion(preds, target)
        noise_loss = -self.log_softmax(preds).sum() / self.num_classes
        return (1 - self.epsilon) * cross_entropy_loss + self.epsilon * noise_loss
```

In [ ]:
```
model = SimpleConvNet_4().to(device)
criterion = LabelSmoothingCrossEntropyLoss(4, weight=torch.Tensor(weights), reduction='mean').to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

model, history, all_time = train(
    model, criterion, optimizer,
    train_loader, val_loader,
    num_epochs=50
)
print(all_time)
```

```
Epoch 50 of 50 took 21.638s
  training loss (in-iteration):         4.535133
  validation loss (in-iteration):       4.053412
  training f1:                  97.40 %
  validation f1:                        90.80 %
```





1161.7567930221558

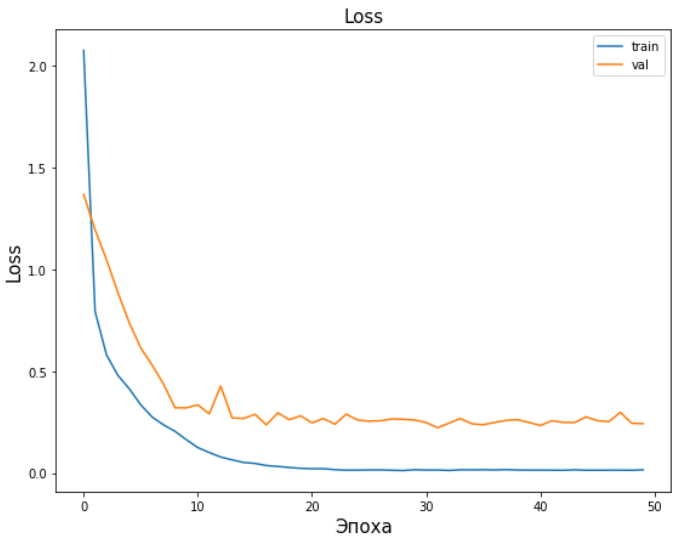
In [ ]:
```
history_model_LS = history.copy()
```

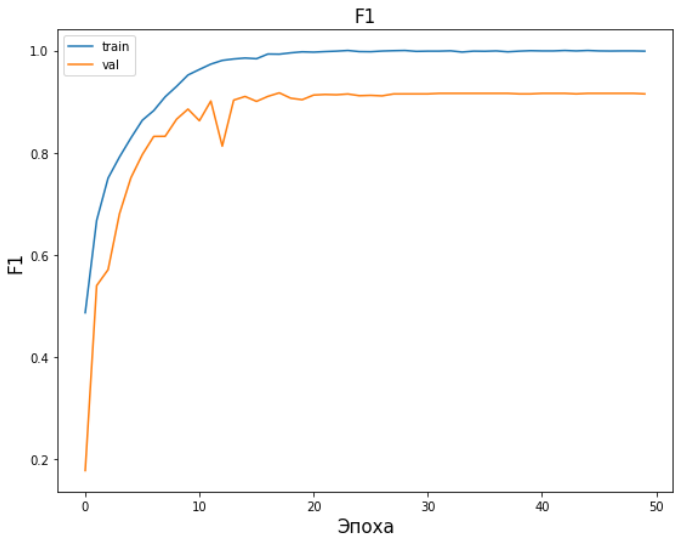
In [ ]:
```
def train_sch(
    model,
    criterion,
    optimizer,
    scheduler,
    train_batch_gen,
    val_batch_gen,
    num_epochs=50
):
    '''
    Функция для обучения модели и вывода лосса и метрики во время обучения.

    :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) F1_score и loss на обучении и валидации ("история" обучения)
    '''
    all_time = 0
    history = defaultdict(lambda: defaultdict(list))

    for epoch in range(num_epochs):
        train_loss = 0
        train_f1 = 0
        train_for_f1_b = []
        train_for_f1_p = []
        val_loss = 0
        val_f1 = 0
        val_for_f1_b = []
        val_for_f1_p = []

        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 = transform_train(X_batch)

            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_for_f1_b = np.append(train_for_f1_b, y_batch.cpu().numpy())
            train_for_f1_p = np.append(train_for_f1_p, y_pred)

        # Подсчитываем лоссы и сохраняем в "историю"
        train_loss /= len(train_batch_gen)
        train_f1 = f1_score(train_for_f1_b, train_for_f1_p, average="macro")
        history['loss']['train'].append(train_loss)
        history['f1']['train'].append(train_f1)

        # 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_for_f1_b = np.append(val_for_f1_b, y_batch.cpu().numpy())
            val_for_f1_p = np.append(val_for_f1_p, y_pred)

        # Подсчитываем лоссы и сохраняем в "историю"
        val_loss /= len(val_batch_gen)
        val_f1 = f1_score(val_for_f1_b, val_for_f1_p, average="macro")
        history['loss']['val'].append(val_loss)
        history['f1']['val'].append(val_f1)

        scheduler.step(val_loss)
        scheduler_steps.append(optimizer.param_groups[0]['lr'])

        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 f1: \t\t\t{:.2f} %".format(train_f1 * 100))
        print("  validation f1: \t\t\t{:.2f} %".format(val_f1 * 100))

        plot_learning_curves(history)

        all_time += (time.time() - start_time)

    return model, history, all_time
```

In [ ]:
```
model = SimpleConvNet_4().to(device)
criterion = nn.CrossEntropyLoss(weight=torch.Tensor(weights), reduction='mean').to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
scheduler = lr_scheduler.ReduceLROnPlateau(optimizer, mode="min", patience=3, factor=0.1)

scheduler_steps = []
model, history, all_time = train_sch(
    model, criterion, optimizer, scheduler,
    train_loader, val_loader,
    num_epochs=50
)
print(all_time)
```

```
Epoch 50 of 50 took 21.204s
  training loss (in-iteration):         0.016011
  validation loss (in-iteration):       0.242547
  training f1:                  99.87 %
  validation f1:                        91.50 %
```

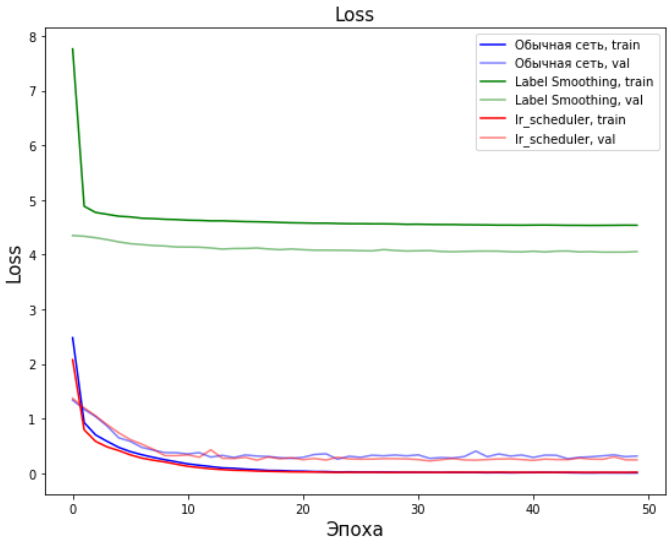


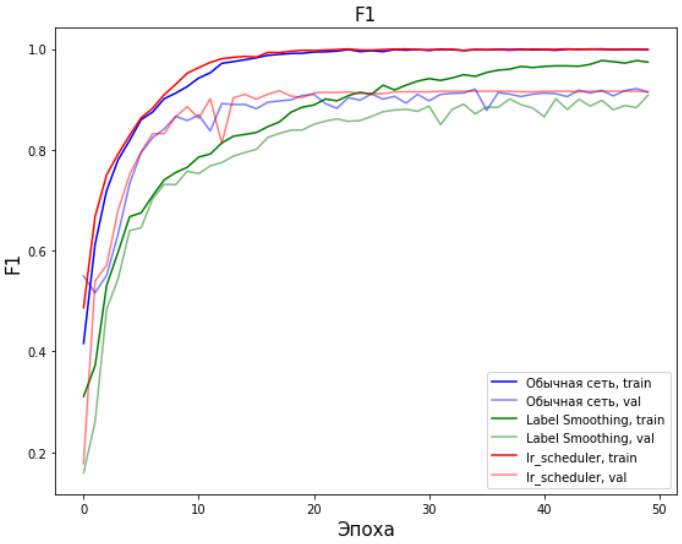



```
238072.92637705803
```

In [ ]:
```
history_model_sch = history.copy()
```

In [ ]:
```
plot_learning_curves_compar([history_model_4, history_model_LS, history_model_sch],
                            ['Обычная сеть', 'Label Smoothing', 'lr_scheduler'])
```





Вывод

Делаем вывод, что у Label Smoothing выше значение лосса и он медленнее, но лучше идет по значенеиям метрики. lr_schedeler пока на таких ранних эпохах не заметен (и у нас стоял изначалльно lr=0.001), но видим, что он достаточно стабилизирует колебания метрики качества на валидации на более поздних эпохах.

Протестируйте своё лучшее решение:

In [ ]:
```
# Лучшее решение – сеть с BatchNorm, Dropout и оптимизатором Adam с lr = 0.001, а также использованием lr_scheduler
# (то есть самая последняя обученная нами модель)
```

In [ ]:
```
# Используйте test_dataset только для финальной оценки качества
test_dataset = torchvision.datasets.ImageFolder(TEST_DIR, transform=transforms.Compose([transforms.ToTensor(),
                                                                                        transforms.Resize((128, 128))
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, shuffle=True)
```

In [ ]:
```
y_b = []
y_p = []
for X_batch, y_batch in test_loader:
    y_b = np.append(y_b, y_batch.cpu().numpy())
    y_p = np.append(y_p, model(X_batch.to(device)).max(1)[1].detach().cpu().numpy())
test_f1_score = f1_score(y_b, y_p, average="macro")
```

In [ ]:
```
print("Итоговый результат:")
print("Test F1 score:\t\t{:.2f} %".format(test_f1_score * 100))
```

```
Итоговый результат:
Test F1 score:          64.77 %
```

Таким образом мы обучили модель, которая спустя 50 эпох обучения на тестовой выборке выдает значение F1-меры около 65% при классфикации вида опухоли по МРТ головного мозга. Это достаточно хороший результат!

Задача 2.

На семинаре вы уже познакомились с основными методами оптимизации, которые широко используются в классическом машинном обучении. С развитием нейронных сетей и активным внедрением нейросетевого подхода, методы оптимизации стали ещё более актуальными. Но стандартные методы оптимизации, SGD и метод тяжёлого шара, имеют ряд недостатков, из-за чего их редко применяют в чистом виде. Для обучения современных нейросетей используют более продвинутые методы.

В данной задаче вам предстоит самостоятельно реализовать различные оптимизаторы (запущенные из одной точки) и сравнить скорости их сходимости.

Пусть задача оптимизации имеет вид $f(x) \longrightarrow \min_x$, и $\nabla_x f(x)$ — градиент функции $f(x)$.

1. SGD

Обычный и стохастический градиентный спуск.

$$x_{t+1} = x_t - \eta v_t,$$

где $v_t = \nabla f(x_t)$ — аналогия со скоростью.

In [ ]:
```python
def sgd(init_parameters, func_grad, lr, n_iter):
    '''
    Метод оптимизации SGD.

    Параметры:
    - parameters - начальное приближение параметров,
    - func_grad - функция, задающая градиент оптимизируемой функции,
    - lr - скорость обучения,
    - n_iter - количество итераций метода.

    Возвращает историю обновлений параметров.
    '''

    parameters = init_parameters.copy()
    history = [parameters.copy()]

    for i in range(n_iter):
        diff = -lr * func_grad(parameters)
        parameters += diff
        history.append(parameters.copy())

    return history
```

2. SGD + Momentum

Сгладим градиент, используя информацию о том, как градиент изменялся раньше. Физическая аналогия — добавляем инерцию.

$$x_{t+1} = x_t + v_t,$$

где $v_t = \mu v_{t-1} - \eta \nabla f(x_t)$ — сглаживаем градиенты.

In [ ]:
```python
def sgd_momentum(init_parameters, func_grad, lr, mu, n_iter):
    '''
    Метод оптимизации SGD Momentum.

    Параметры:
    - parameters - начальное приближение параметров,
    - func_grad - функция, задающая градиент оптимизируемой функции,
    - lr - скорость обучения,
    - mu - коэффициент сглаживания,
    - n_iter - количество итераций метода.

    Возвращает историю обновлений параметров.
    '''
    parameters = init_parameters.copy()
    history = [parameters.copy()]
    diff = np.zeros_like(parameters)

    for i in range(n_iter):
        diff = mu * diff - lr * func_grad(parameters)
        parameters += diff
        history.append(parameters.copy())

    return history
```

3. Adagrad

Adagrad — один из самых первых адаптивных методов оптимизации.

Во всех изученных ранее методах есть необходимость подбирать шаг метода (коэффициент η). На каждой итерации все компоненты градиента оптимизируемой функции домножаются на одно и то же число η. Но использовать одно значение η для всех параметров не оптимально, так как они имеют различные распределения и оптимизируемая функция изменяется с совершенно разной скоростью при небольших изменениях разных параметров.

Поэтому гораздо логичнее **изменять значение каждого параметра с индивидуальной скоростью**. При этом, *чем с большей степени от изменения параметра меняется значение оптимизируемой функции, тем с меньшей скоростью стоить обновлять этот параметр*. Иначе высок шанс расходимости метода. Получить такой результат удается, если разделить градиент на сумму квадратов скорости изменений параметров.

Пусть $x^{(i)}$ — i-я компонента вектора x.

$$x_{t+1,i} = x_{t,i} - \frac{\eta}{\sqrt{g_{t,i} + \varepsilon}} \cdot \nabla f_i(x_t)$$

$$g_t = g_{t-1} + \nabla f(x_t) \odot \nabla f(x_t)$$

В матрично-векторном виде шаг алгоритма можно переписать так:

$$x_{t+1} = x_t - \frac{\eta}{\sqrt{g_t + \varepsilon}} \odot \nabla f(x_t).$$

Здесь $\odot$ обозначает произведение Адамара, т.е. поэлементное перемножение векторов.

In [ ]:
```python
def adagrad(init_parameters, func_grad, lr, eps, n_iter):
    '''
    Метод оптимизации Adagrad.

    Параметры:
    - parameters - начальное приближение параметров,
    - func_grad - функция, задающая градиент оптимизируемой функции,
    - lr - скорость обучения,
    - eps - минимальное значение нормирующего члена,
    - n_iter - количество итераций метода.

    Возвращает историю обновлений параметров.
    '''
    parameters = init_parameters.copy()
    history = [parameters.copy()]
    norm = np.zeros_like(parameters)

    for iter_id in range(n_iter):
        cur_grad = func_grad(parameters)
        norm += cur_grad ** 2
        parameters -= lr * cur_grad / np.sqrt(norm + eps)
        history.append(parameters.copy())

    return history
```

4. RMSProp

Алгоритм RMSProp основан на той же идее, что и алгоритм Adagrad — адаптировать learning rate отдельно для каждого параметра $\theta^{(i)}$. Однако Adagrad имеет серьёзный недостаток. Он с одинаковым весом учитывает квадраты градиентов как с самых первых итераций, так и с самых последних. Хотя, на самом деле, наибольшую значимость имеют модули градиентов на последних нескольких итерациях.

Для этого предлагается использовать **экспоненциальное сглаживание**.

$$x_{t+1} = x_t - \frac{\eta}{\sqrt{g_t + \varepsilon}} \odot \nabla f(x_t).$$

$$g_t = \mu g_{t-1} + (1 - \mu) \nabla f(x_t) \odot \nabla f(x_t)$$

Стандартные значения гиперпараметров: $\mu = 0.9, \eta = 0.001$.

In [ ]:
```python
def rmsprop(init_parameters, func_grad, lr, mu, eps, n_iter):
    '''
    Метод оптимизации RMSProp.

    Параметры:
    - parameters - начальное приближение параметров,
    - func_grad - функция, задающая градиент оптимизируемой функции,
    - lr - скорость обучения,
    - mu - коэффициент сглаживания,
    - eps - минимальное значение нормирующего члена,
    - n_iter - количество итераций метода.

    Возвращает историю обновлений параметров.
    '''

    parameters = init_parameters.copy()
```

```
    history = [parameters.copy()]
    norm = np.zeros_like(parameters)

    for iter_id in range(n_iter):
        cur_grad = func_grad(parameters)
        norm = mu * norm + (1 - mu) * cur_grad ** 2
        parameters -= lr * cur_grad / np.sqrt(norm + eps)
        history.append(parameters.copy())

    return history
```

5. Adam

Этот алгоритм совмещает в себе 2 идеи:

- идею алгоритма Momentum *о накапливании градиента*,
- идею методов Adadelta и RMSProp *об экспоненциальном сглаживании* информации о предыдущих значениях квадратов градиентов.

Благодаря использованию этих двух идей, метод имеет 2 преимущества над большей частью методов первого порядка, описанных выше:

1. Он обновляет все параметры θ не с одинаковым `learning rate`, а выбирает для каждого θ_i индивидуальный `learning rate`, что *позволяет учитывать разреженные признаки с большим весом*.

2. Adam за счёт применения экспоненциального сглаживания к градиенту *работает более устойчиво в окрестности оптимального значения параметра $|theta^*, чемметоды, использующиеградиентвточкех_t$, не накапливая значения градиента с прошлых шагов.

Формулы шага алгоритма выглядят так:

$$v_{t+1} = \beta v_t + (1 - \beta)\nabla x(x_t)$$

$$g_t = \mu g_{t-1} + (1 - \mu)\nabla x(x_t) \odot \nabla x(x_t)$$

Чтобы эти оценки не были смещёнными, нужно их отнормировать:

$$\hat{v}_{t+1} = \frac{v_{t+1}}{1 - \beta^t},$$

$$\hat{g}_t = \frac{g_t}{1 - \mu^t}.$$

Тогда получим итоговую формулу шага:

$$x_{t+1} = x_t - \frac{\eta}{\sqrt{\hat{g}_t + \varepsilon}} \odot \hat{v}_{t+1}.$$

In [ ]:
```
def adam(init_parameters, func_grad, eps, lr, beta, mu,  n_iter):
    '''
    Adam.

    Параметры.
    1) theta0 - начальное приближение theta,
    2) func_grad - функция, задающая градиент оптимизируемой функции,
    3) eps - мин. возможное значение знаменателя,
    4) lr - скорость обучения,
    5) beta - параметр экспоненциального сглаживания,
    6) mu - параметр экспоненциального сглаживания,
    7) n_iter - количество итераций метода.
    '''

    parameters = init_parameters.copy()
    history = [parameters.copy()]
    cumulative_m = np.zeros_like(parameters)
    cumulative_v = np.zeros_like(parameters)
    pow_beta, pow_mu = beta, mu

    for iter_id in range(n_iter):
        current_grad = func_grad(parameters)
        cumulative_m = beta * cumulative_m + (1 - beta) * current_grad
        cumulative_v = mu * cumulative_v + (1 - mu) * current_grad ** 2

        scaled_m = cumulative_m / (1 - pow_beta)
        scaled_v = cumulative_v / (1 - pow_mu)
        parameters = parameters - lr * cumulative_m / (np.sqrt(cumulative_v) + eps)
        history.append(parameters)

        pow_beta *= beta
        pow_mu *= mu

    return history
```

Сравнение оптимизаторов будем производить на примере двух функций:

1. $f(x, y) = 5 * x^2 + y^2$
2. $f(x, y) = (x - 3)^2 + 8(y - 5)^4 + \sqrt{x} + \sin(xy)$

Реализуйте данные функции в ячейке ниже для удобства и читаемости кода.

In [ ]:
```
def square_sum(x):
    ''' f(x, y) = 5 * x^2 + y^2 '''

    return 5 * x[0]**2 + x[1]**2

def square_sum_grad(x):
    ''' grad f(x, y) = (10x, 2y) '''

    return np.array([10, 2]) * x


def complex_sum(x):
    ''' f(x, y) = (x-3)^2 + 8(y-5)^4 + sqrt(x) + sin(xy)'''

    return (x[0]-3)**2 + 8 * (x[1]-5)**4 + x[0]**0.5 + np.sin(x[0]*x[1])

def complex_sum_grad(x):
    ''' grad f(x, y) = (2(x-3) + 1/(2 sqrt(x)) + ycos(xy), 32(y-5)^3 + xcos(xy)) '''

    partial_x = 2*(x[0]-3) + 0.5*x[0]**(-0.5) + x[1]*np.cos(x[0]*x[1])
    partial_y = 32*(x[1]-5)**3 + x[0]*np.cos(x[0]*x[1])

    return np.array([partial_x, partial_y])
```

Создадим директорию, в которой будем хранить визуализацию экспериментов.

In [ ]:
```
!rm -rf saved_gifs
!mkdir saved_gifs
```

```
"rm" Ґ пў«пҐвбп ўгваҐҐ© Ё«Ё ўҐиҐ©
Є®¬ ¤®©, ЁбЇ®«пҐ¬®© Їа®Ја ¬¬®© Ё«Ё Ї ЄҐвл¬ д ©«®¬.
```

Напишем функцию, которая будет отрисовывать процесс оптимизации.

In [ ]:
```
def make_experiment(func, trajectory, graph_title,
                    min_y=-7, max_y=7, min_x=-7, max_x=7):
    '''
    Функция, которая для заданной функции рисует её линии уровня,
    а также траекторию сходимости метода оптимизации.

    Параметры.
    1) func - оптимизируемая функция,
    2) trajectory - траектория метода оптимизации,
    3) graph_name - заголовок графика.
    '''

    fig, ax = plt.subplots(figsize=(10, 8))
    xdata, ydata = [], []
    ln, = plt.plot([], [])

    mesh_x = np.linspace(min_x, max_x, 300)
    mesh_y = np.linspace(min_y, max_y, 300)
    X, Y = np.meshgrid(mesh_x, mesh_y)
    Z = np.zeros((len(mesh_x), len(mesh_y)))

    for coord_x in range(len(mesh_x)):
        for coord_y in range(len(mesh_y)):
            Z[coord_y, coord_x] = func(
                np.array((mesh_x[coord_x],
                          mesh_y[coord_y]))
            )

    def init():
        ax.contour(
            X, Y, np.log(Z),
            np.log([0.5, 10, 30, 80, 130, 200, 300, 500, 900]),
            cmap='winter'
        )
        ax.set_title(graph_title)
        return ln,

    def update(frame):
        xdata.append(trajectory[frame][0])
        ydata.append(trajectory[frame][1])
        ln.set_data(xdata, ydata)
        return ln,

    ani = FuncAnimation(
```

```
        fig, update, frames=range(len(trajectory)),
        init_func=init, repeat=True
    )
    writer = ImageMagickFileWriter(fps=10)
    ani.save(f'saved_gifs/{graph_title}.gif',
             writer=writer)
    plt.show()
```

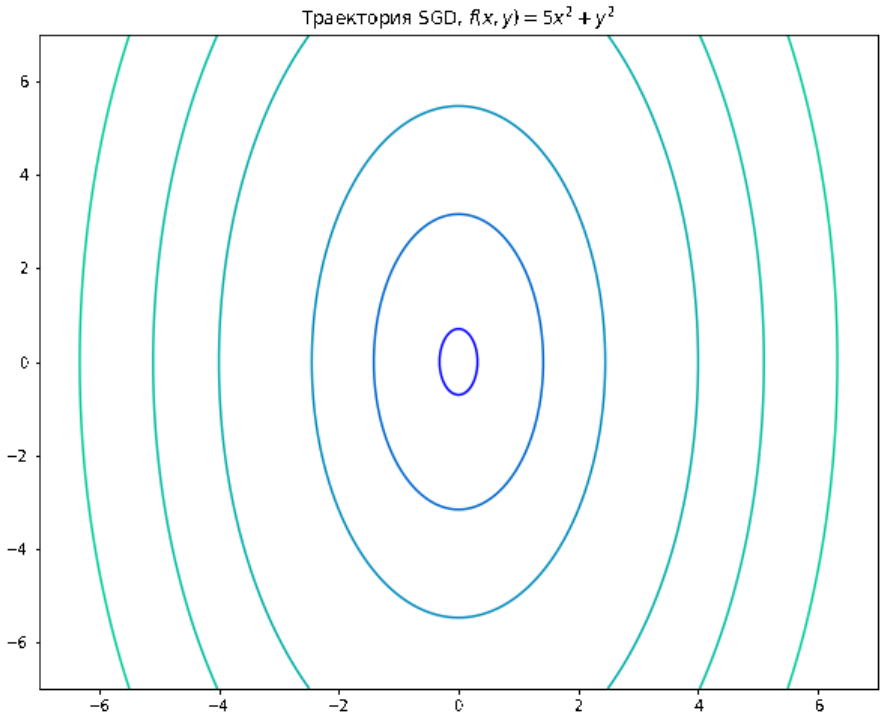
Простая функция $f(x, y) = x^2 + y^2$

In [ ]:
```
parameters = np.array((5, 5), dtype=float)
func_name = '$f(x, y) = 5x^2 + y^2$'
func_grad = square_sum_grad
func = square_sum
n_iter = 100
```

SGD

In [ ]:
```
sgd_trajectory = sgd(
    init_parameters=parameters,
    func_grad=func_grad,
    lr=0.01,
    n_iter=n_iter
)
graph_title = 'Траектория SGD, ' + func_name
make_experiment(
    func,
    sgd_trajectory,
    graph_title,
)
clear_output()
Image(open(f'saved_gifs/{graph_title}.gif','rb').read())
```

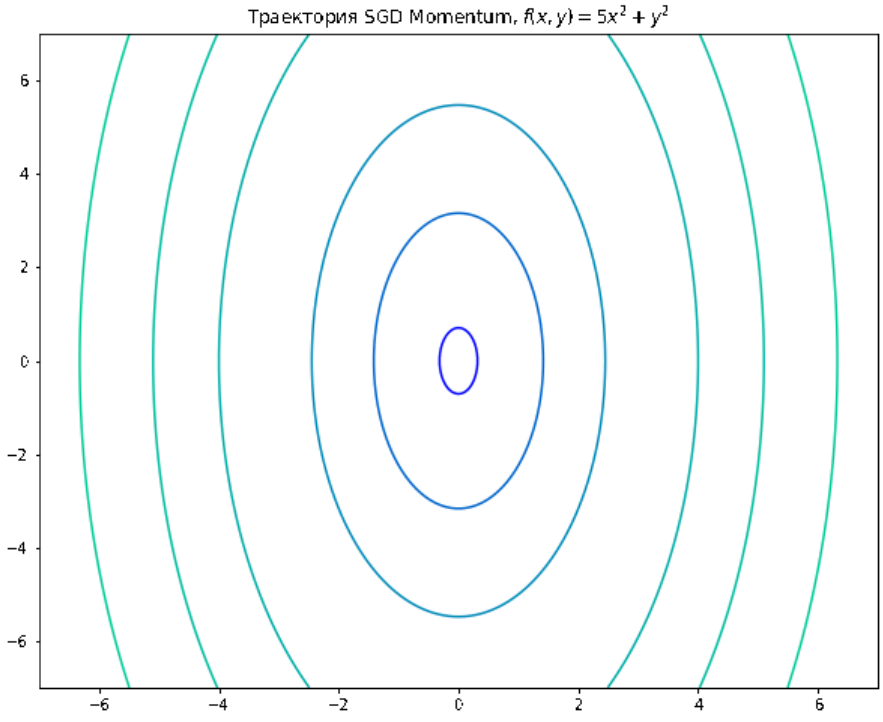
Out[ ]:



SGD Momentum

In [ ]:
```
sgd_momentum_trajectory = sgd_momentum(
    init_parameters=parameters.copy(),
    func_grad=func_grad,
    lr=0.01,
    n_iter=n_iter,
    mu=0.9
)
graph_title = 'Траектория SGD Momentum, ' + func_name
make_experiment(
    func,
    sgd_momentum_trajectory,
    graph_title,
)
clear_output()
Image(open(f'saved_gifs/{graph_title}.gif','rb').read())
```

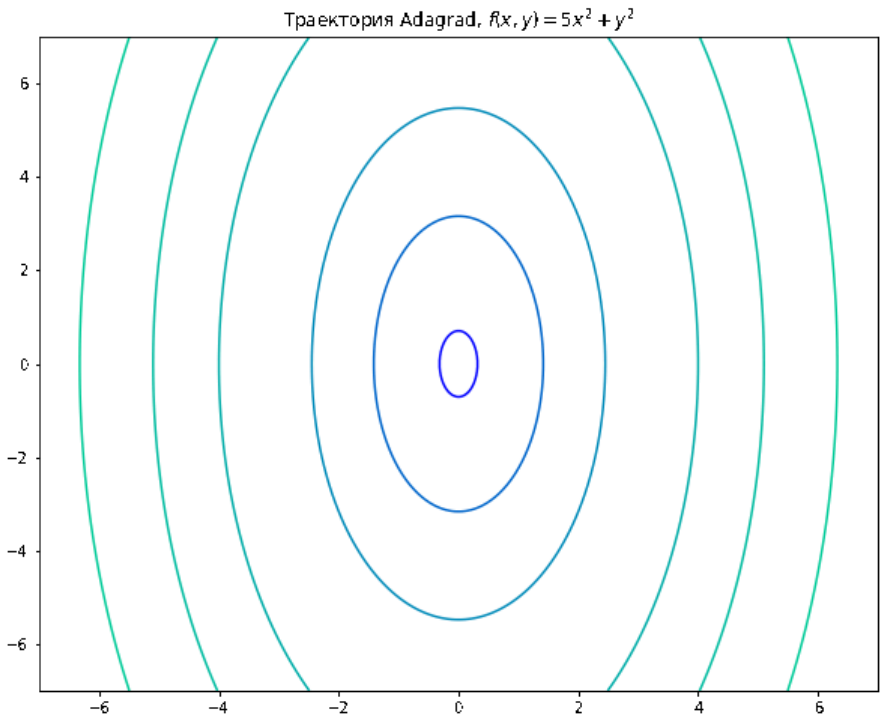
Out[ ]:



Adagrad

In [ ]:
```
adagrad_trajectory = adagrad(
    init_parameters=parameters.copy(),
    func_grad=func_grad,
    lr=0.1,
    n_iter=n_iter,
    eps=1e-6,
)
graph_title = 'Траектория Adagrad, ' + func_name
make_experiment(
    func,
    adagrad_trajectory,
    graph_title,
)
clear_output()
Image(open(f'saved_gifs/{graph_title}.gif','rb').read())
```

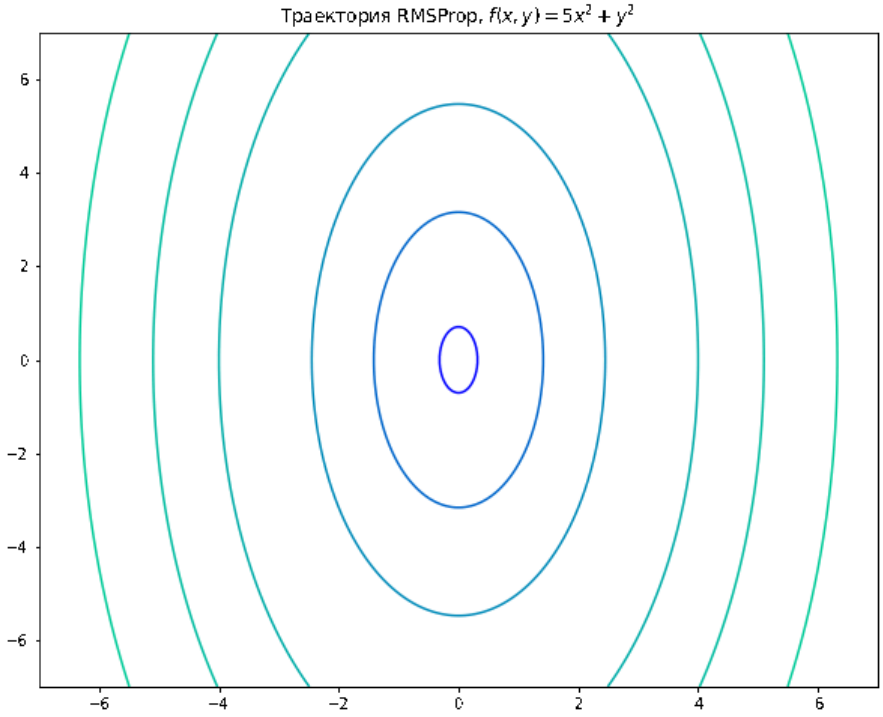
Out[ ]:



RMSProp

In [ ]:
```
rmsprop_trajectory = rmsprop(
    init_parameters=parameters.copy(),
    func_grad=func_grad,
    lr=0.1,
    n_iter=n_iter,
    eps=1e-6,
    mu=0.9
)
graph_title = 'Траектория RMSProp, ' + func_name
make_experiment(
    func,
    rmsprop_trajectory,
    graph_title,
)
clear_output()
Image(open(f'saved_gifs/{graph_title}.gif','rb').read())
```

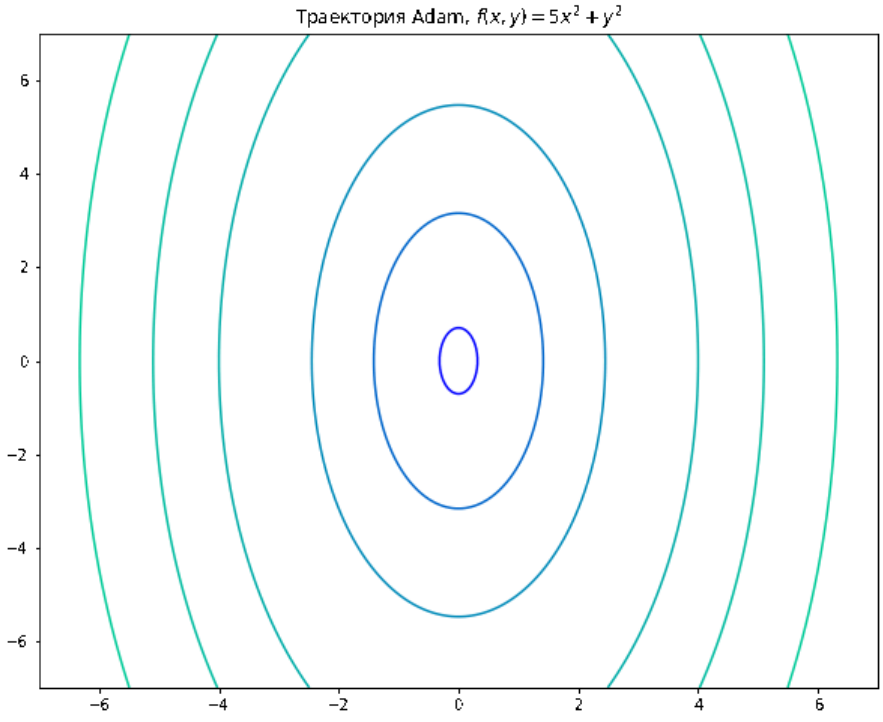
Out[ ]:



Adam

In [ ]:
```
adam_trajectory = adam(
    init_parameters=parameters.copy(),
    func_grad=func_grad,
    lr=0.1,
    n_iter=n_iter,
    eps=1e-6,
    mu=0.9,
    beta=0.9
)
graph_title = 'Траектория Adam, ' + func_name
make_experiment(
    func,
    adam_trajectory,
    graph_title,
)
clear_output()
Image(open(f'saved_gifs/{graph_title}.gif','rb').read())
```

Out[ ]:



Вывод

Замечаем, что SGD медленно, но довольно хорошо идет в сторону минимума. SGD + Momentum делает это быстрее и более хаотично, но в итоге приходит в район минимума. Adagrad, RMSprop и Adam изначально идут в минимум по прямому маршруту, но с разной скоростью. Adagrad медленнее всего, а Adam - быстрее.

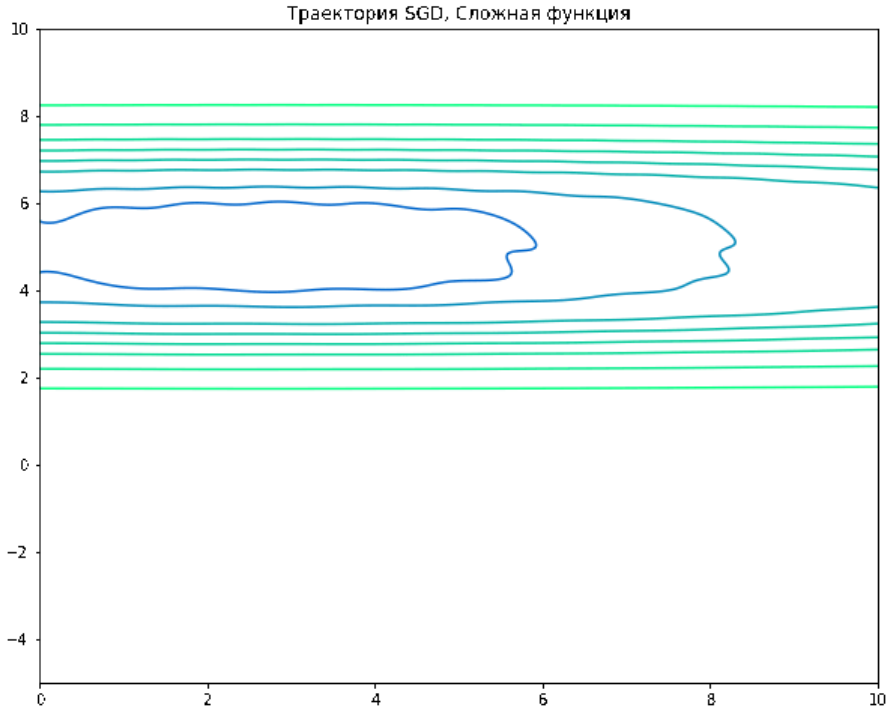
Сложная функция $f(x, y) = (x - 3)^2 + 8(y - 5)^4 + \sqrt{x} + \sin(xy)$

In [ ]:
```
parameters = np.array((5, -2), dtype=float)
func_name = 'Сложная функция'#'$f(x, y) = (x-3)^2 + 8(y-5)^4 + \sqrt{x} + \sin(xy)$'
func_grad = complex_sum_grad
func = complex_sum
n_iter = 100
```

SGD

In [ ]:
```
sgd_trajectory = sgd(
    init_parameters=parameters,
    func_grad=func_grad,
    lr=0.0002,
    n_iter=n_iter
)
graph_title = 'Траектория SGD, ' +  func_name
make_experiment(
    func,
    sgd_trajectory,
    graph_title,
    -5, 10, 0, 10
)
clear_output()
Image(open(f'saved_gifs/{graph_title}.gif','rb').read())
```

Out[ ]:

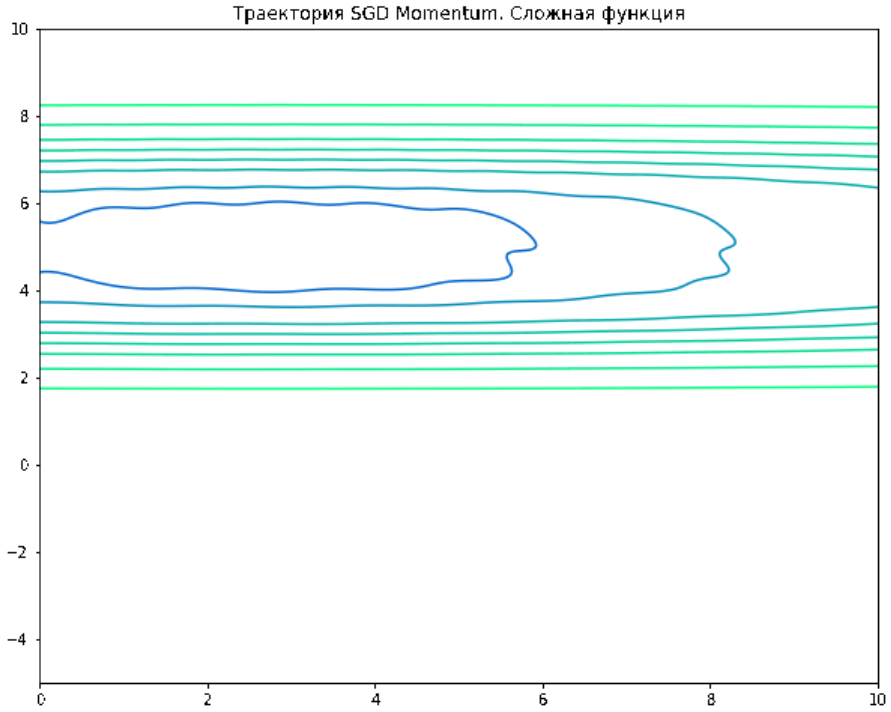


SGD Momentum

In [ ]:

```
sgd_momentum_trajectory = sgd_momentum(
    init_parameters=parameters.copy(),
    func_grad=func_grad,
    lr=0.0002,
    n_iter=n_iter,
    mu=0.7
)
graph_title = 'Траектория SGD Momentum, ' +  func_name
make_experiment(
    func,
    sgd_momentum_trajectory,
    graph_title,
    -5, 10, 0, 10
)
clear_output()
Image(open(f'saved_gifs/{graph_title}.gif','rb').read())
```

Out[ ]:

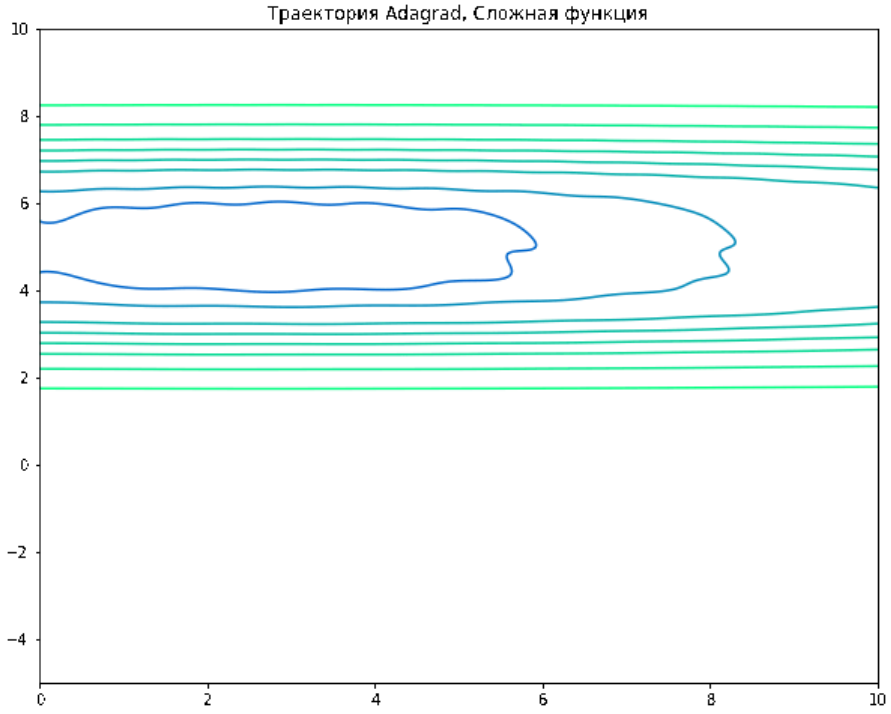


Adagrad

In [ ]:

```
adagrad_trajectory = adagrad(
    init_parameters=parameters.copy(),
    func_grad=func_grad,
    lr=0.1,
    n_iter=n_iter,
    eps=1e-6,
)
graph_title = 'Траектория Adagrad, ' + func_name
make_experiment(
    func,
    adagrad_trajectory,
    graph_title,
    -5, 10, 0, 10
)
clear_output()
Image(open(f'saved_gifs/{graph_title}.gif','rb').read())
```

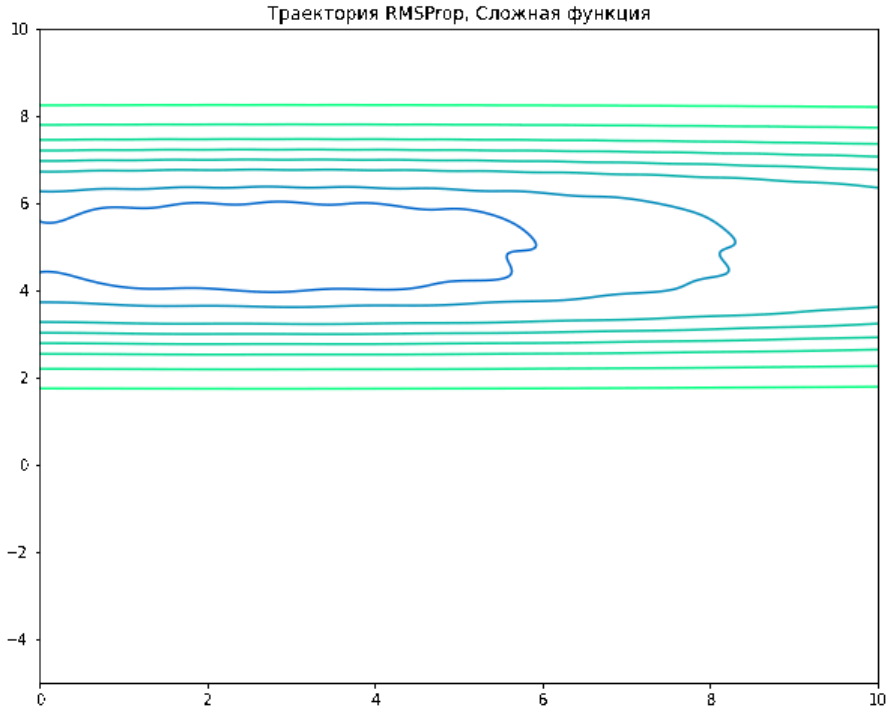
Out[ ]:



RMSProp

In [ ]:
```
rmsprop_trajectory = rmsprop(
    init_parameters=parameters.copy(),
    func_grad=func_grad,
    lr=0.1,
    n_iter=n_iter,
    eps=1e-6,
    mu=0.9
)
graph_title = 'Траектория RMSProp, ' + func_name
make_experiment(
    func,
    rmsprop_trajectory,
    graph_title,
    -5, 10, 0, 10
)
clear_output()
Image(open(f'saved_gifs/{graph_title}.gif','rb').read())
```

Out[ ]:

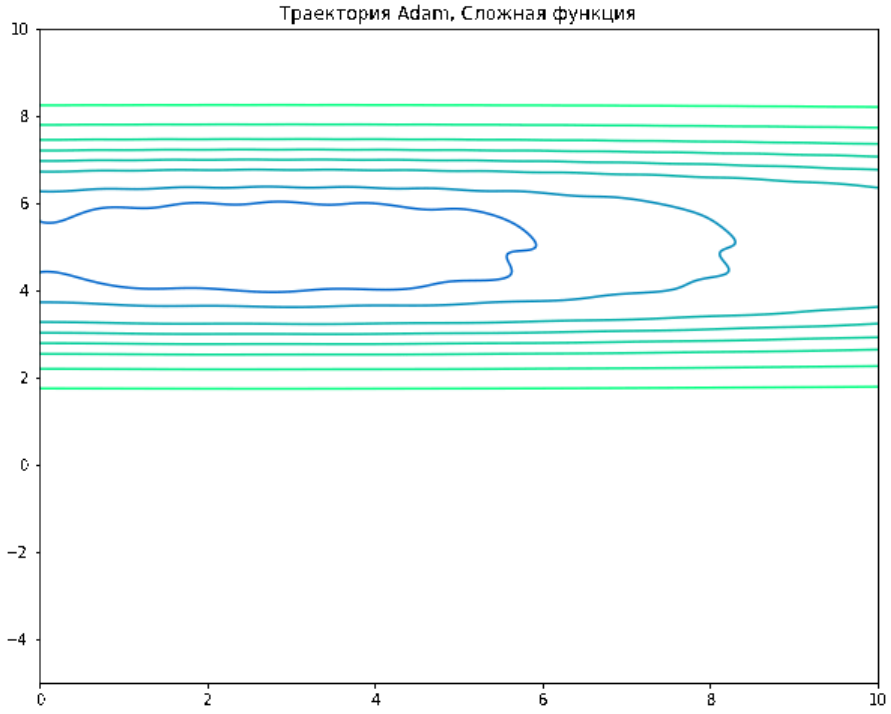


Adam

In [ ]:

```
adam_trajectory = adam(
    init_parameters=parameters.copy(),
    func_grad=func_grad,
    lr=0.1,
    n_iter=n_iter,
    eps=1e-6,
    mu=0.9,
    beta=0.9
)
graph_title = 'Траектория Adam, ' + func_name
make_experiment(
    func,
    adam_trajectory,
    graph_title,
    -5, 10, 0, 10
)
clear_output()
Image(open(f'saved_gifs/{graph_title}.gif','rb').read())
```

Out[ ]:



Вывод

В случае сложной функции мы видим, что SGD вновь довольно медленно, но ровно идет в сторону отрицательного градтента, SGD+Momentem делает это быстрее. Adagrad, RMSprop и Adam идут примерно по одной траектории. Но при этом Adagrad движется очень медленно и даже не приближается к минимуму. Adam приближается быстрее остальных и делает это более плавно, чем RMSprop.