

Практическое задание №1

Установка необходимых пакетов:

```
!pip install -q tqdm
!pip install --upgrade --no-cache-dir gdown

Requirement already satisfied: gdown in /usr/local/lib/python3.10/dist-packages (4.7.1)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from gdown) (3.13.1)
Requirement already satisfied: requests[socks] in /usr/local/lib/python3.10/dist-packages (from gdown) (2.31.0)
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages (from gdown) (1.16.0)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from gdown) (4.66.1)
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.10/dist-packages (from gdown) (4.11.2)
Requirement already satisfied: soupsieve>1.2 in /usr/local/lib/python3.10/dist-packages (from beautifulsoup4->gdown) (2.5)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests[socks]->gdown) (3)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests[socks]->gdown) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests[socks]->gdown) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests[socks]->gdown) (2023.7)
Requirement already satisfied: PySocks!=1.5.7,>=1.5.6 in /usr/local/lib/python3.10/dist-packages (from requests[socks]->gdown) (1.7
```

Монтирование Вашего Google Drive к текущему окружению:

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

Mounted at /content/drive
```

Константы, которые пригодятся в коде далее, и ссылки (gdrive идентификаторы) на предоставляемые наборы данных:

```
EVALUATE_ONLY = False
TEST_ON_LARGE_DATASET = True
TISSUE_CLASSES = ('ADI', 'BACK', 'DEB', 'LYM', 'MUC', 'MUS', 'NORM', 'STR', 'TUM')
DATASETS_LINKS = {
    'train': '1XtQzVQ5XbrfxpLHJuL0XBGJ5U7CS-cLi',
    'local_train': '1e4GeqU0uIlbJ9_ENJ6mM13kUGwuasuvz',
    'local_test': '1jU1N5Ki9ikwdJMAaqy7xhVMz2ZKNkzWj_',
    'train_small': '1qd45xXfDwdZjktLFwQb-et-mAaFeCzOR',
    'train_tiny': '1I-2Z0uXLd4QwhZQQLtp817Kn3J0Xgbui',
    'test': '1RfPou3pFKpuHDJZ-D9XDFzgvpUBF1Dr',
    'test_small': '1wbRsog0n7uG1HIPGLhyN-PMET2kdQ2LI',
    'test_tiny': '1viiB0s041CNsAK4itvX8PnYthJ-MDnQc'
}
```

Импорт необходимых зависимостей:

```
from pathlib import Path
import numpy as np
from typing import List
from tqdm.notebook import tqdm
from time import sleep
from PIL import Image
import IPython.display
from sklearn.metrics import balanced_accuracy_score, confusion_matrix
import gdown

import time

import torch
from torch import nn
from torch.nn import functional as F
import torchvision
from torchvision.transforms import v2

import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns
import warnings; warnings.filterwarnings(action='once')

large = 22; med = 16; small = 12
params = {'axes.titlesize': large,
          'legend.fontsize': med,
          'figure.figsize': (16, 10),
          'axes.labelsize': med,
```

```

        'axes.titlesize': med,
        'xtick.labelsize': med,
        'ytick.labelsize': med,
        'figure.titlesize': large}
plt.rcParams.update(params)
plt.style.use('seaborn-whitegrid')
sns.set_style("white")
%matplotlib inline

```

```

<ipython-input-4-d75d84943b53>:34: MatplotlibDeprecationWarning: The seaborn styles shipped by Matplotlib are deprecated since 3.6,
plt.style.use('seaborn-whitegrid')

```

▼ Класс Dataset

Предназначен для работы с наборами данных, обеспечивает чтение изображений и соответствующих меток, а также формирование пакетов (батчей).

```
class Dataset:
```

```

    def __init__(self, name):
        self.name = name
        self.is_loaded = False
        url = f"https://drive.google.com/uc?export=download&confirm=pbef&id={DATASETS_LINKS[name]}"
        output = f'{name}.npz'
        gdown.download(url, output, quiet=False)
        print(f'Loading dataset {self.name} from npz.')
        np_obj = np.load(f'{name}.npz')
        self.images = np_obj['data']
        self.labels = np_obj['labels']
        self.n_files = self.images.shape[0]
        self.is_loaded = True
        print(f'Done. Dataset {name} consists of {self.n_files} images.')

```

```

    def image(self, i):
        # read i-th image in dataset and return it as numpy array
        if self.is_loaded:
            return self.images[i, :, :, :]

```

```

    def images_seq(self, n=None):
        # sequential access to images inside dataset (is needed for testing)
        for i in range(self.n_files if not n else n):
            yield self.image(i)

```

```

    def random_image_with_label(self):
        # get random image with label from dataset
        i = np.random.randint(self.n_files)
        return self.image(i), self.labels[i]

```

```

    def random_batch_with_labels(self, n):
        # create random batch of images with labels (is needed for training)
        indices = np.random.choice(self.n_files, n)
        imgs = []
        for i in indices:
            img = self.image(i)
            imgs.append(self.image(i))
        logits = np.array([self.labels[i] for i in indices])
        return np.stack(imgs), logits

```

```

    def image_with_label(self, i: int):
        # return i-th image with label from dataset
        return self.image(i), self.labels[i]

```

```

/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_
and should_run_async(code)

```

▼ Пример использования класса Dataset

Загрузим обучающий набор данных, получим произвольное изображение с меткой. После чего визуализируем изображение, выведем метку. В будущем, этот кусок кода можно закомментировать или убрать.

```

d_train_tiny = Dataset('train_tiny')

img, lbl = d_train_tiny.random_image_with_label()
print()
print(f'Got numpy array of shape {img.shape}, and label with code {lbl}.')

```

```
print(f'Label code corresponds to {TISSUE_CLASSES[1b1]} class.')
```

```
pil_img = Image.fromarray(img)
IPython.display.display(pil_img)
```

▼ Класс Metrics

Реализует метрики точности, используемые для оценивания модели:

1. точность,
2. сбалансированную точность.

```
class Metrics:

    @staticmethod
    def accuracy(gt: List[int], pred: List[int]):
        assert len(gt) == len(pred), 'gt and prediction should be of equal length'
        return sum(int(i[0] == i[1]) for i in zip(gt, pred)) / len(gt)

    @staticmethod
    def accuracy_balanced(gt: List[int], pred: List[int]):
        return balanced_accuracy_score(gt, pred)

    @staticmethod
    def print_all(gt: List[int], pred: List[int], info: str):
        print(f'metrics for {info}:')
        print('\t accuracy {:.4f}'.format(Metrics.accuracy(gt, pred)))
        print('\t balanced accuracy {:.4f}'.format(Metrics.accuracy_balanced(gt, pred)))
```

▼ Класс Model

Класс, хранящий в себе всю информацию о модели.

Вам необходимо реализовать методы `save`, `load` для сохранения и загрузки модели. Особенно актуально это будет во время тестирования на дополнительных наборах данных.

Пожалуйста, убедитесь, что сохранение и загрузка модели работает корректно. Для этого обучите модель, протестируйте, сохраните ее в файл, перезапустите среду выполнения, загрузите обученную модель из файла, вновь протестируйте ее на тестовой выборке и убедитесь в том, что получаемые метрики совпадают с полученными для тестовой выборки ранее.

Также, Вы можете реализовать дополнительные функции, такие как:

1. валидацию модели на части обучающей выборки;
2. использование кроссвалидации;
3. автоматическое сохранение модели при обучении;
4. загрузку модели с какой-то конкретной итерации обучения (если используется итеративное обучение);
5. вывод различных показателей в процессе обучения (например, значение функции потерь на каждой эпохе);
6. построение графиков, визуализирующих процесс обучения (например, график зависимости функции потерь от номера эпохи обучения);
7. автоматическое тестирование на тестовом наборе/наборах данных после каждой эпохи обучения (при использовании итеративного обучения);
8. автоматический выбор гиперпараметров модели во время обучения;
9. сохранение и визуализацию результатов тестирования;
10. Использование аугментации и других способов синтетического расширения набора данных (дополнительным плюсом будет обоснование необходимости и обоснование выбора конкретных типов аугментации)
11. и т.д.

Полный список опций и дополнений приведен в презентации с описанием задания.

При реализации дополнительных функций допускается добавление параметров в существующие методы и добавление новых методов в класс модели.

```
def kfold_split(num_objects: int,
                num_folds: int) -> list[tuple[np.ndarray, np.ndarray]]:
    step = num_objects // num_folds
    ans = []
    for i in range(0, num_folds - 1):
        ans.append((np.concatenate((np.arange(0, i * step, dtype=np.int32),
                                     np.arange((i + 1) * step, num_objects, dtype=np.int32))), np.arange(i * step, (i + 1) * step, dtype=
        ans.append((np.arange(0, (num_folds - 1) * step, dtype=np.int32), np.arange((num_folds - 1) * step, num_objects, dtype=np.int32)))
```

```

return ans

class Dataloader:
    ''' Класс для загрузки батчей из датасета:
    1) С применением необходимых для torch преобразований
    2) С применением преобразований генерирующих дополнительные данные
    3) Есть возможность использовать часть тестовой выборки (если нужна частая валидация)
    '''
    def __init__(self, dataset : Dataset, indices : np.array, batch_size, transforms=None, augmenting_transforms=None, limit=1.):
        self.dataset = dataset
        self.indices = indices
        self.batch_size = batch_size
        self.num_batch = len(indices) // batch_size
        self.n_batch = -1
        self.transforms = transforms
        self.augmenting_transforms = augmenting_transforms
        self.limit = limit

    def __iter__(self):
        self.n_batch = 0
        return self

    def __next__(self):

        np.random.shuffle(self.indices)

        if self.n_batch < self.num_batch * self.limit:

            first = self.n_batch * self.batch_size
            images = torch.empty((self.batch_size, 3, 224, 224))
            labels = torch.empty(self.batch_size, dtype=torch.int32)

            for i in range(self.batch_size):
                image, label = self.dataset.image_with_label(self.indices[first + i])
                image = torch.from_numpy(image).permute(2, 0, 1)
                #L6
                if self.augmenting_transforms:
                    image = self.augmenting_transforms(image)
                if self.transforms:
                    image = self.transforms(image)
                images[i] = image
                labels[i] = torch.tensor(label)

            self.n_batch += 1
            return [images.type(torch.FloatTensor), labels.type(torch.LongTensor)]
        else:
            raise StopIteration

class CrossVal:
    #L6
    def __init__(self, dataset : Dataset, batch_size, num_folds, transforms=None, augmenting_transforms=None, seed=42, limit=(1., 1.)):
        self.dataset = dataset
        self.batch_size = batch_size
        self.num_folds = num_folds
        self.folds = kfold_split(dataset.n_files, num_folds)
        self.indices = np.arange(0, dataset.n_files)
        self.transforms = transforms
        self.augmenting_transforms = augmenting_transforms
        np.random.seed(seed)
        np.random.shuffle(self.indices)
        self.n_val = 0
        self.limit = limit

    def get_train_val(self):
        train_indices = self.indices[self.folds[self.n_val][0]]
        val_indices = self.indices[self.folds[self.n_val][1]]
        self.n_val = (self.n_val + 1) % self.num_folds
        return (Dataloader(self.dataset, train_indices, self.batch_size, self.transforms, self.augmenting_transforms, limit=self.limit[0],
                           Dataloader(self.dataset, val_indices, self.batch_size, self.transforms, self.augmenting_transforms, limit=self.limit[1])

def create_model(model, num_freeze_layers, num_out_classes):
    model.fc = nn.Linear(512, num_out_classes)

    for i, layer in enumerate(model.children()):
        if i < num_freeze_layers:
            for param in layer.parameters():
                param.requires_grad = False

    return model

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

```

```

def plot_train_process(train_loss, val_loss, train_accuracy, val_accuracy, title_suffix=''):
    fig, axes = plt.subplots(1, 2, figsize=(15, 5))

    axes[0].set_title(' '.join(['Loss', title_suffix]))
    axes[0].plot(train_loss, label='train')
    axes[0].plot(val_loss, label='validation')
    axes[0].legend()

    axes[1].set_title(' '.join(['Validation accuracy', title_suffix]))
    axes[1].plot(train_accuracy, label='train')
    axes[1].plot(val_accuracy, label='validation')
    axes[1].legend()
    plt.show()

class Model:

    def __init__(self):
        self.model = create_model(torchvision.models.resnet18(pretrained=True), 6, 9).to(device)
        #self.transforms = torchvision.models.ResNet18_Weights.IMAGENET1K_V1.transforms()
        self.transforms = v2.Compose([
            v2.ToDtype(torch.float32, scale=True),
            v2.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),
        ])

    def save(self, name: str):
        torch.save(self.model.state_dict(), f'/content/drive/MyDrive/Digital Pathology/{name}.txt')
        #np.savez(f'/content/drive/MyDrive/{name}.npz', data=arr)

    def load(self, name: str):
        name_to_id_dict = {
            'best': '1RY0ja4fM77nmqxLUrXJb-F0qd6R9I4A3'
        }
        output = f'{name}.npz'
        gdown.download(f'https://drive.google.com/uc?id={name_to_id_dict[name]}', output, quiet=False)
        self.model.load_state_dict(torch.load(output))
        self.model.to(device)
        self.model.eval()

    def evaluate(self, dataloader : DataLoader, loss_fn, weights):

        losses = []

        num_correct = np.zeros(9, dtype=int)
        num_elements = np.zeros(9, dtype=int)

        for i, batch in enumerate(dataloader):

            # так получаем текущий батч
            X_batch, y_batch = batch

            with torch.no_grad():
                logits = self.model(X_batch.to(device))

                loss = loss_fn(logits, y_batch.to(device))
                losses.append(loss.item())

                y_pred = torch.argmax(logits, dim=1).cpu()
                num_elements += (y_batch.numpy()[None, :].T.repeat(9, axis=1) == np.arange(9)).astype(int).sum(axis=0)

                num_correct += ((y_batch.numpy()[None, :].T.repeat(9, axis=1) == np.arange(9)).astype(int) * (y_batch.numpy()[None, :].

        acc = num_correct / num_elements
        #LBL7
        weights = torch.from_numpy(acc ** 6)

        accuracy = num_correct.sum() / num_elements.sum()

        return accuracy, np.mean(losses), weights

    def train(self, dataset: Dataset):

        optim = torch.optim.Adamax
        lossF = torch.nn.CrossEntropyLoss

        learning_rate = 1e-4

        optimizer = optim(self.model.parameters(), lr=learning_rate)
        #LBL8
        scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=5, threshold=0.01, threshold_mode='rel')

        #LBL6

```

```

augmenting_transforms = v2.Compose([
    v2.RandomHorizontalFlip(),
    v2.RandomVerticalFlip(),
    #v2.ColorJitter(brightness=0.12, contrast=0.3)
])

batch_size = 32
folds = 10
limit = (0.3, 1.)
dataloader = CrossVal(dataset, batch_size, folds, self.transforms, augmenting_transforms=augmenting_transforms, limit=limit)
print(f'training started')

history = ''

train_loss_history = []
train_acc_history = []
val_loss_history = []
val_acc_history = []

local_train_loss_history = []
local_train_acc_history = []

n_epoch = 40
eval_every = 40

weights = torch.ones(9)

for epoch in range(n_epoch):

    print("Epoch:", epoch+1)
    history += f"Epoch: {epoch+1}\n"
    #LBL7
    loss_fn = lossF(weight=weights.type(torch.FloatTensor).to(device))

    train, val = dataloader.get_train_val()

    self.model.train(True)

    for i, batch in enumerate(train):

        start_time = time.time()
        X_batch, y_batch = batch

        logits = self.model(X_batch.to(device))

        loss = loss_fn(logits, y_batch.to(device))

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

        model_answers = torch.argmax(logits, dim=1)
        train_accuracy = torch.sum(y_batch == model_answers.cpu()) / len(y_batch)

        local_train_loss_history.append(loss.item())
        local_train_acc_history.append(train_accuracy)

        if (i + 1) % eval_every == 0:
            #LBL3
            history += f"Средние train лосс и accuracy на последних {eval_every} итерациях: {np.mean(local_train_loss_history)}\n"
            print(f"Средние train лосс и accuracy на последних {eval_every} итерациях:",
                  np.mean(local_train_loss_history), np.mean(local_train_acc_history), end='\n')

    self.model.train(False)

    #LBL1
    val_accuracy, val_loss, weights = self.evaluate(val, loss_fn, weights)
    #LBL8
    scheduler.step(val_loss)

    train_loss_history.append(np.mean(local_train_loss_history))
    train_acc_history.append(np.mean(local_train_acc_history))
    val_loss_history.append(val_loss)
    val_acc_history.append(val_accuracy)

    IPython.display.clear_output(wait=True)
    #LBL4
    plot_train_process(train_loss_history, val_loss_history, train_acc_history, val_acc_history)

    history += f"Эпоха {epoch+1}/{n_epoch}: val лосс и accuracy: {val_loss} {val_accuracy} \n"
    history += f"New weights {weights} \n"
    history += f"Learning rate {optimizer.state_dict()['param_groups'][0]['lr']} \n"

```

```

        #self.save(f'Epoch {epoch+1} Val accuracy {val_accuracy}')
        #LBL3
        print(history)

    print(f'training done')

def test_on_dataset(self, dataset: Dataset, limit=None):
    # you can upgrade this code if you want to speed up testing using batches
    predictions = []
    n = dataset.n_files if not limit else int(dataset.n_files * limit)
    for img in tqdm(dataset.images_seq(n), total=n):
        predictions.append(self.test_on_image(img))
    return predictions

def test_on_image(self, img: np.ndarray):
    prediction = self.model(self.transforms(torch.from_numpy(img).permute(2, 0, 1))[None, :, :, :].to(device))
    prediction = int(torch.argmax(prediction, dim=1).cpu()[0])
    return prediction

```

▼ Классификация изображений

Используя введенные выше классы можем перейти уже непосредственно к обучению модели классификации изображений. Пример общего пайплайна решения задачи приведен ниже. Вы можете его расширять и улучшать. В данном примере используются наборы данных 'train_small' и 'test_small'.

```

#d_train = Dataset('train_small')
d_train = Dataset('local_train')

```

```

Downloading...
From: https://drive.google.com/uc?export=download&confirm=pbef&id=1e4GegU0uIlbJ9\_ENJ6mM13kUGwuasuvz
To: /content/local_train.npz
100%|██████████| 2.10G/2.10G [00:23<00:00, 91.1MB/s]
Loading dataset local_train from npz.
Done. Dataset local_train consists of 18000 images.

```

```
model = Model()
```

```

/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated sin
warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None`
warnings.warn(msg)

```

```
model.load('best')
```

```

Downloading...
From: https://drive.google.com/uc?id=1RY0ja4fM77nmqxLUrXJb-FQqd6R9I4A3
To: /content/best.npz
100%|██████████| 44.8M/44.8M [00:00<00:00, 186MB/s]

```

```

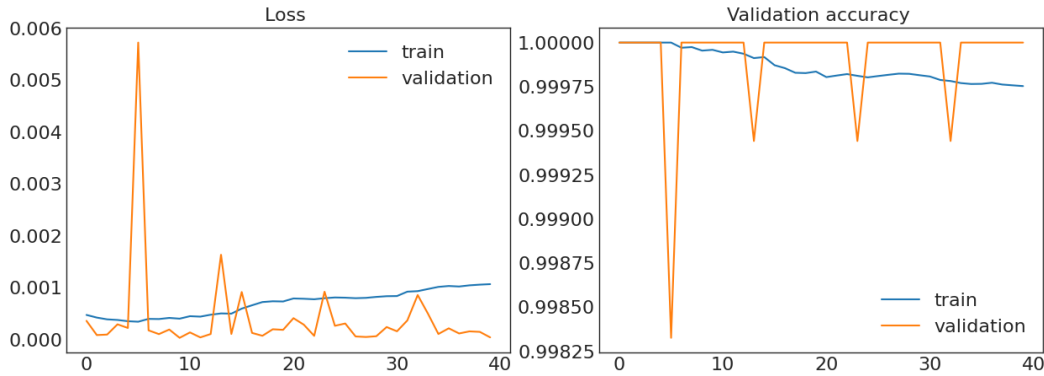
for i, layer in enumerate(model.model.children()):
    if i < 3:
        for param in layer.parameters():
            param.requires_grad = False
    else:
        for param in layer.parameters():
            param.requires_grad = True

```

```

if not EVALUATE_ONLY:
    model.train(d_train)
    model.save('best')
else:
    #todo: your link goes here
    model.load('best')

```



```
Epoch: 1
Средние train loss и accuracy на последних 40 итерациях: 0.0003929504780629145 1.0
Средние train loss и accuracy на последних 40 итерациях: 0.0004660609637859636 1.0
Средние train loss и accuracy на последних 40 итерациях: 0.00048215150088471624 1.0
Эпоха 1/40: val loss и accuracy: 0.00035682546824286225 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 0.0001
Epoch: 2
Средние train loss и accuracy на последних 40 итерациях: 0.0005217661510158086 1.0
Средние train loss и accuracy на последних 40 итерациях: 0.00047875581951497626 1.0
Средние train loss и accuracy на последних 40 итерациях: 0.0004435255765686296 1.0
Эпоха 2/40: val loss и accuracy: 8.233068752667399e-05 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 0.0001
Epoch: 3
Средние train loss и accuracy на последних 40 итерациях: 0.00043955145811959835 1.0
Средние train loss и accuracy на последних 40 итерациях: 0.00040979442524665305 1.0
Средние train loss и accuracy на последних 40 итерациях: 0.00040560376904366803 1.0
Эпоха 3/40: val loss и accuracy: 9.218671364205225e-05 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 0.0001
Epoch: 4
Средние train loss и accuracy на последних 40 итерациях: 0.00036538572826927407 1.0
Средние train loss и accuracy на последних 40 итерациях: 0.00035340330530834167 1.0
Средние train loss и accuracy на последних 40 итерациях: 0.00035805721370183206 1.0
Эпоха 4/40: val loss и accuracy: 0.0002893906176016766 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 0.0001
Epoch: 5
Средние train loss и accuracy на последних 40 итерациях: 0.0003643758096958794 1.0
Средние train loss и accuracy на последних 40 итерациях: 0.0003522856253074273 1.0
Средние train loss и accuracy на последних 40 итерациях: 0.0003441473685001185 1.0
Эпоха 5/40: val loss и accuracy: 0.00021900121422951152 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 0.0001
Epoch: 6
Средние train loss и accuracy на последних 40 итерациях: 0.0003430594601302772 1.0
Средние train loss и accuracy на последних 40 итерациях: 0.00034176187200521163 1.0
Средние train loss и accuracy на последних 40 итерациях: 0.0003360514966329669 1.0
Эпоха 6/40: val loss и accuracy: 0.005718601423295817 0.9983258928571429
New weights tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.9445, 0.9678, 1.0000, 1.0000],
dtype=torch.float64)
Learning rate 0.0001
Epoch: 7
Средние train loss и accuracy на последних 40 итерациях: 0.0004055950809487882 0.9999671578407288
Средние train loss и accuracy на последних 40 итерациях: 0.00039949092134554165 0.9999684691429138
Средние train loss и accuracy на последних 40 итерациях: 0.00040261738569062416 0.9999697208404541
Эпоха 7/40: val loss и accuracy: 0.00017294244476114988 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 0.0001
Epoch: 8
Средние train loss и accuracy на последних 40 итерациях: 0.0004044608138422603 0.9999716877937317
Средние train loss и accuracy на последних 40 итерациях: 0.0003988536364556754 0.9999727010726929
Средние train loss и accuracy на последних 40 итерациях: 0.0003939696067460618 0.9999735951423645
Эпоха 8/40: val loss и accuracy: 0.00010062946122754315 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1e-05
Epoch: 9
Средние train loss и accuracy на последних 40 итерациях: 0.00038887344307020875 0.9999751448631287
Средние train loss и accuracy на последних 40 итерациях: 0.00042145346751512293 0.9999517798423767
Средние train loss и accuracy на последних 40 итерациях: 0.00041479212162698277 0.9999532103538513
Эпоха 9/40: val loss и accuracy: 0.00018894899320441385 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1e-05
Epoch: 10
Средние train loss и accuracy на последних 40 итерациях: 0.0004057866801828635 0.9999555945396423
Средние train loss и accuracy на последних 40 итерациях: 0.00040200216143836527 0.9999568462371826
Средние train loss и accuracy на последних 40 итерациях: 0.00040346592709830144 0.9999579787254333
Эпоха 10/40: val loss и accuracy: 2.906511723769053e-05 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1e-05
Epoch: 11
Средние train loss и accuracy на последних 40 итерациях: 0.0003928595157835829 0.9999599456787109
Средние train loss и accuracy на последних 40 итерациях: 0.0004066388788456166 0.9999600680676731
```



```
Средние train loss и accuracy на последних 40 итерациях: 0.0004000200/004301020 0.9999000000/0121
Средние train loss и accuracy на последних 40 итерациях: 0.0004145508634611768 0.9999619126319885
Эпоха 11/40: val loss и accuracy: 0.000133192058081631 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1e-05
Epoch: 12
Средние train loss и accuracy на последних 40 итерациях: 0.00043761991688783116 0.9999452233314514
Средние train loss и accuracy на последних 40 итерациях: 0.0004353796556713714 0.9999464750289917
Средние train loss и accuracy на последних 40 итерациях: 0.00043740560470070404 0.9999476671218872
Эпоха 12/40: val loss и accuracy: 3.7224235370051216e-05 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1e-05
Epoch: 13
Средние train loss и accuracy на последних 40 итерациях: 0.00047618988361623456 0.9999329447746277
Средние train loss и accuracy на последних 40 итерациях: 0.00046935084021029397 0.9999343752861023
Средние train loss и accuracy на последних 40 итерациях: 0.0004717436893318824 0.9999356865882874
Эпоха 13/40: val loss и accuracy: 0.00010243222952206159 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1e-05
Epoch: 14
Средние train loss и accuracy на последних 40 итерациях: 0.0005035624048989492 0.9999070167541504
Средние train loss и accuracy на последних 40 итерациях: 0.0005006880029230765 0.9999088048934937
Средние train loss и accuracy на последних 40 итерациях: 0.000498283797274284 0.9999105334281921
Эпоха 14/40: val loss и accuracy: 0.0016296424710051025 0.9994419642857143
New weights tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.9724, 1.0000],
dtype=torch.float64)
Learning rate 1e-05
Epoch: 15
Средние train loss и accuracy на последних 40 итерациях: 0.0005026896136681586 0.9999135136604309
Средние train loss и accuracy на последних 40 итерациях: 0.0005027575722413728 0.9999150633811951
Средние train loss и accuracy на последних 40 итерациях: 0.0004988030458325894 0.9999166131019592
Эпоха 15/40: val loss и accuracy: 0.00010340799839728529 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1e-05
Epoch: 16
Средние train loss и accuracy на последних 40 итерациях: 0.0005024127141982492 0.9999057054519653
Средние train loss и accuracy на последних 40 итерациях: 0.0005003141022441074 0.9999073147773743
Средние train loss и accuracy на последних 40 итерациях: 0.00051737686194614 0.9998828172683716
Эпоха 16/40: val loss и accuracy: 0.000911860243364109 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000002e-06
Epoch: 17
Средние train loss и accuracy на последних 40 итерациях: 0.0005987188297034496 0.999860942363739
Средние train loss и accuracy на последних 40 итерациях: 0.0006139808565930313 0.9998631477355957
Средние train loss и accuracy на последних 40 итерациях: 0.0006459448753010492 0.9998530745506287
Эпоха 17/40: val loss и accuracy: 0.0001236672858908605 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000002e-06
Epoch: 18
Средние train loss и accuracy на последних 40 итерациях: 0.0007132879275251734 0.9998332858085632
Средние train loss и accuracy на последних 40 итерациях: 0.0007229956243846856 0.999824047088623
Средние train loss и accuracy на последних 40 итерациях: 0.0007221384744399657 0.9998266696929932
Эпоха 18/40: val loss и accuracy: 6.843664481185183e-05 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000002e-06
Epoch: 19
Средние train loss и accuracy на последних 40 итерациях: 0.0007313697422760675 0.9998198747634888
Средние train loss и accuracy на последних 40 итерациях: 0.000738645578038246 0.9998224377632141
Средние train loss и accuracy на последних 40 итерациях: 0.0007382464838413853 0.9998249411582947
Эпоха 19/40: val loss и accuracy: 0.00019404715267004998 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000002e-06
Epoch: 20
Средние train loss и accuracy на последних 40 итерациях: 0.0007331784235188994 0.9998292326927185
Средние train loss и accuracy на последних 40 итерациях: 0.0007263567947928568 0.9998315572738647
Средние train loss и accuracy на последних 40 итерациях: 0.0007229214949034449 0.9998337626457214
Эпоха 20/40: val loss и accuracy: 0.00018323300363525568 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000002e-06
Epoch: 21
Средние train loss и accuracy на последних 40 итерациях: 0.000735656477862857 0.99982750415802
Средние train loss и accuracy на последних 40 итерациях: 0.0007319302889906673 0.9998297095298767
Средние train loss и accuracy на последних 40 итерациях: 0.0007592349912040081 0.9998220205307007
Эпоха 21/40: val loss и accuracy: 0.0004079188890579693 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000002e-06
Epoch: 22
Средние train loss и accuracy на последних 40 итерациях: 0.0007867447671135601 0.9998066425323486
Средние train loss и accuracy на последних 40 итерациях: 0.0007894353378051774 0.9998089671134949
Средние train loss и accuracy на последних 40 итерациях: 0.0007859214931622089 0.9998112916946411
Эпоха 22/40: val loss и accuracy: 0.0002821671108027398 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000002e-07
Epoch: 23
Средние train loss и accuracy на последних 40 итерациях: 0.000778428754038683 0.9998152852058411
Средние train loss и accuracy на последних 40 итерациях: 0.0007720613060174911 0.9998174905776978
Средние train loss и accuracy на последних 40 итерациях: 0.000774158848283226 0.9998195767402649
Эпоха 23/40: val loss и accuracy: 6.689424073361547e-05 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000002e-07
Epoch: 24
```

```
Средние train loss и accuracy на последних 40 итерациях: 0.0007653854785411527 0.999823272228241
Средние train loss и accuracy на последних 40 итерациях: 0.0007648462864593926 0.9998252391815186
Средние train loss и accuracy на последних 40 итерациях: 0.0007601750421940999 0.9998271465301514
Эпоха 24/40: val loss и accuracy: 0.0009176886372220647 0.9994419642857143
New weights tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.9749, 1.0000],
dtype=torch.float64)
Learning rate 1.0000000000000002e-07
Epoch: 25
Средние train loss и accuracy на последних 40 итерациях: 0.000790426519360808 0.9998135566711426
Средние train loss и accuracy на последних 40 итерациях: 0.0008133395431566872 0.9997988343238831
Средние train loss и accuracy на последних 40 итерациях: 0.000810303268604919 0.999800980091095
Эпоха 25/40: val loss и accuracy: 0.000260795078215129 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000002e-07
Epoch: 26
Средние train loss и accuracy на последних 40 итерациях: 0.0008081147747438511 0.999804675579071
Средние train loss и accuracy на последних 40 итерациях: 0.0008042090355361506 0.9998067021369934
Средние train loss и accuracy на последних 40 итерациях: 0.000803835023778175 0.999808669090271
Эпоха 26/40: val loss и accuracy: 0.000305283967059771 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000002e-07
Epoch: 27
Средние train loss и accuracy на последних 40 итерациях: 0.0007995361818790609 0.999812126159668
Средние train loss и accuracy на последних 40 итерациях: 0.0007949126480877452 0.999813973903656
Средние train loss и accuracy на последних 40 итерациях: 0.000796958546643254 0.999815821647644
Эпоха 27/40: val loss и accuracy: 5.542415365310051e-05 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000002e-07
Epoch: 28
Средние train loss и accuracy на последних 40 итерациях: 0.0007955995492092 0.9998190402984619
Средние train loss и accuracy на последних 40 итерациях: 0.0007989499920877275 0.9998207688331604
Средние train loss и accuracy на последних 40 итерациях: 0.0007987181072202491 0.9998224377632141
Эпоха 28/40: val loss и accuracy: 4.5164905916042895e-05 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000004e-08
Epoch: 29
Средние train loss и accuracy на последних 40 итерациях: 0.0008215997340534665 0.9998181462287903
Средние train loss и accuracy на последних 40 итерациях: 0.000822453776337727 0.999819815158844
Средние train loss и accuracy на последних 40 итерациях: 0.0008190417628385863 0.9998214840888977
Эпоха 29/40: val loss и accuracy: 5.985496472454039e-05 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000004e-08
Epoch: 30
Средние train loss и accuracy на последних 40 итерациях: 0.000817407799899082 0.9998243451118469
Средние train loss и accuracy на последних 40 итерациях: 0.0008138795907043661 0.9998259544372559
Средние train loss и accuracy на последних 40 итерациях: 0.0008289923203660803 0.9998136758804321
Эпоха 30/40: val loss и accuracy: 0.00023696825053016419 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000004e-08
Epoch: 31
Средние train loss и accuracy на последних 40 итерациях: 0.0008396658037831875 0.9998030066490173
Средние train loss и accuracy на последних 40 итерациях: 0.0008373481402370429 0.999804675579071
Средние train loss и accuracy на последних 40 итерациях: 0.0008336116514391728 0.9998063445091248
Эпоха 31/40: val loss и accuracy: 0.00015474564981080415 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000004e-08
Epoch: 32
Средние train loss и accuracy на последних 40 итерациях: 0.000837967691416898 0.9998092651367188
Средние train loss и accuracy на последних 40 итерациях: 0.0009057868557249727 0.9997913241386414
Средние train loss и accuracy на последних 40 итерациях: 0.0009195499602999047 0.9997865557670593
Эпоха 32/40: val loss и accuracy: 0.00036456961603091386 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000004e-08
Epoch: 33
Средние train loss и accuracy на последних 40 итерациях: 0.0009212747186645264 0.9997833371162415
Средние train loss и accuracy на последних 40 итерациях: 0.0009316088818829893 0.9997787475585938
Средние train loss и accuracy на последних 40 итерациях: 0.0009281751895614972 0.999780535697937
Эпоха 33/40: val loss и accuracy: 0.0008537232977993929 0.9994419642857143
New weights tensor([1.0000, 1.0000, 1.0000, 1.0000, 0.9739, 1.0000, 1.0000, 1.0000, 1.0000],
dtype=torch.float64)
Learning rate 1.0000000000000004e-08
Epoch: 34
Средние train loss и accuracy на последних 40 итерациях: 0.0009252339142820228 0.9997836947441101
Средние train loss и accuracy на последних 40 итерациях: 0.0009316954142803701 0.999779224395752
Средние train loss и accuracy на последних 40 итерациях: 0.0009366885234755162 0.9997748732566833
Эпоха 34/40: val loss и accuracy: 0.0004915811932961438 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000004e-08
Epoch: 35
Средние train loss и accuracy на последних 40 итерациях: 0.0009832355092691761 0.9997659921646118
Средние train loss и accuracy на последних 40 итерациях: 0.0010001721010548634 0.9997618198394775
Средние train loss и accuracy на последних 40 итерациях: 0.001014030770162489 0.9997636079788208
Эпоха 35/40: val loss и accuracy: 0.00010306823250364314 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000004e-08
Epoch: 36
Средние train loss и accuracy на последних 40 итерациях: 0.0010097636420576467 0.9997667670249939
Средние train loss и accuracy на последних 40 итерациях: 0.00102821454828857 0.9997627139091492
Средние train loss и accuracy на последних 40 итерациях: 0.0010274410201193377 0.9997645020484924
Эпоха 36/40: val loss и accuracy: 0.0002124783429420875 1.0
```

```

New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000004e-08
Epoch: 37
Средние train loss и accuracy на последних 40 итерациях: 0.0010259384128771683 0.999767541885376
Средние train loss и accuracy на последних 40 итерациях: 0.0010257006021628936 0.9997692108154297
Средние train loss и accuracy на последних 40 итерациях: 0.0010222433132490107 0.9997708797454834
Эпоха 37/40: val loss и accuracy: 0.00011318600342186918 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000004e-08
Epoch: 38
Средние train loss и accuracy на последних 40 итерациях: 0.0010231379862809336 0.9997682571411133
Средние train loss и accuracy на последних 40 итерациях: 0.001018591651949356 0.999769926071167
Средние train loss и accuracy на последних 40 итерациях: 0.0010327456956349735 0.9997660517692566
Эпоха 38/40: val loss и accuracy: 0.0001535437744126154 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000004e-08
Epoch: 39
Средние train loss и accuracy на последних 40 итерациях: 0.0010603995362531602 0.9997528195381165
Средние train loss и accuracy на последних 40 итерациях: 0.0010598914836646763 0.9997545480728149
Средние train loss и accuracy на последних 40 итерациях: 0.0010556088513157082 0.9997562170028687
Эпоха 39/40: val loss и accuracy: 0.0001440839464313472 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000004e-08
Epoch: 40
Средние train loss и accuracy на последних 40 итерациях: 0.0010583348898465942 0.9997538924217224
Средние train loss и accuracy на последних 40 итерациях: 0.0010675647548154654 0.9997503161430359
Средние train loss и accuracy на последних 40 итерациях: 0.0010641154995120106 0.9997519850730896
Эпоха 40/40: val loss и accuracy: 3.7863305015507975e-05 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=torch.float64)
Learning rate 1.0000000000000004e-08

#LBLE
def conf_matrix(gt: List[int], pred: List[int]):
    plt.figure(figsize=(8,6), dpi= 80)
    matrix = confusion_matrix(gt, pred)
    sns.heatmap(matrix, xticklabels=np.arange(9), yticklabels=np.arange(9), cmap='RdYlGn', center=0, annot=True)

    # Decorations
    plt.title('Confusion Matrix', fontsize=22)
    plt.xticks(fontsize=12)
    plt.yticks(fontsize=12)
    plt.show()

```

Пример тестирования модели на части набора данных:

```

# evaluating model on 10% of test dataset
pred_1 = model.test_on_dataset(d_test, limit=0.1)
conf_matrix(d_test.labels[:len(pred_1)], pred_1)
Metrics.print_all(d_test.labels[:len(pred_1)], pred_1, '10% of test')

d_real_test = Dataset('local_test')

Downloading...
From: https://drive.google.com/uc?export=download&confirm=pbef&id=1jU1N5Ki9ikwdJMAagy7xhVMz2ZNKzWj\_
To: /content/local_test.npz
100%|██████████| 525M/525M [00:05<00:00, 100MB/s]
Loading dataset local_test from npz.
Done. Dataset local_test consists of 4500 images.

```

Пример тестирования модели на полном наборе данных:

```


# evaluating model on full test dataset (may take time)
pred_2 = None
if TEST_ON_LARGE_DATASET:
    pred_2 = model.test_on_dataset(d_real_test)
    Metrics.print_all(d_real_test.labels, pred_2, 'test')

/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning
and should_run_async(code)

100% 4500/4500 [00:26<00:00, 195.97it/s]

metrics for test:
    accuracy 0.9940:
    balanced accuracy 0.9940:

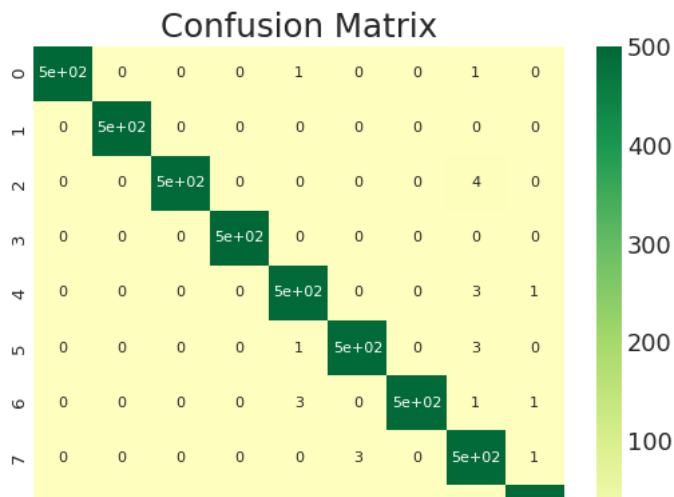
```

◀  ▶

```
conf_matrix(d_real_test.labels, pred_2)
```



```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning
and should_run_async(code)
```



Результат работы пайплайна обучения и тестирования выше тоже будет оцениваться. Поэтому не забудьте присылать на проверку ноутбук с выполненными ячейками кода с демонстрациями метрик обучения, графиками и т.п. В этом пайплайне Вам необходимо продемонстрировать работу всех реализованных дополнений, улучшений и т.п.

Настоятельно рекомендуется после получения пайплайна с полными результатами обучения экспортировать ноутбук в pdf (файл -> печать) и прислать этот pdf вместе с самим ноутбуком.

▼ Тестирование модели на других наборах данных

Ваша модель должна поддерживать тестирование на других наборах данных. Для удобства, Вам предоставляется набор данных `test_tiny`, который представляет собой малую часть (2% изображений) набора `test`. Ниже приведен фрагмент кода, который будет осуществлять тестирование для оценивания Вашей модели на дополнительных тестовых наборах данных.

Прежде чем отсылать задание на проверку, убедитесь в работоспособности фрагмента кода ниже.

```
final_model = Model()
final_model.load('best')
d_test_tiny = Dataset('test_tiny')
pred = final_model.test_on_dataset(d_test_tiny)
Metrics.print_all(d_test_tiny.labels, pred, 'test-tiny')
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'p
warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other
warnings.warn(msg)
Downloading...
From: https://drive.google.com/uc?id=1RY0ja4fM77nmqXLUrXJb-F0qd6R9I4A3
To: /content/best.npz
100%|██████████| 44.8M/44.8M [00:00<00:00, 195MB/s]
Downloading...
From: https://drive.google.com/uc?export=download&confirm=pbef&id=1viiB0s041CNsAK4itvX8PnYthJ-MDnQc
To: /content/test_tiny.npz
100%|██████████| 10.6M/10.6M [00:00<00:00, 201MB/s]Loading dataset test_tiny from npz.
Done. Dataset test_tiny consists of 90 images.
```

```
100% 90/90 [00:00<00:00, 174.07it/s]

metrics for test-tiny:
  accuracy 0.9889:
  balanced accuracy 0.9889:
```

Отмонтировать Google Drive.

```
drive.flush_and_unmount()
```

▼ Дополнительные "полезности"

Ниже приведены примеры использования различных функций и библиотек, которые могут быть полезны при выполнении данного практического задания.

▼ Измерение времени работы кода

Измерять время работы какой-либо функции можно легко и непринужденно при помощи функции `timeit` из соответствующего модуля:

```
import timeit

def factorial(n):
    res = 1
    for i in range(1, n + 1):
        res *= i
    return res

def f():
    return factorial(n=1000)

n_runs = 128
print(f'Function f is caluclated {n_runs} times in {timeit.timeit(f, number=n_runs)}s.')
```

▼ Scikit-learn

Для использования "классических" алгоритмов машинного обучения рекомендуется использовать библиотеку `scikit-learn` (<https://scikit-learn.org/stable/>). Пример классификации изображений цифр из набора данных MNIST при помощи классификатора SVM:

```
# Standard scientific Python imports
import matplotlib.pyplot as plt

# Import datasets, classifiers and performance metrics
from sklearn import datasets, svm, metrics
from sklearn.model_selection import train_test_split

# The digits dataset
digits = datasets.load_digits()

# The data that we are interested in is made of 8x8 images of digits, let's
# have a look at the first 4 images, stored in the `images` attribute of the
# dataset. If we were working from image files, we could load them using
# matplotlib.pyplot.imread. Note that each image must have the same size. For these
# images, we know which digit they represent: it is given in the 'target' of
# the dataset.
_, axes = plt.subplots(2, 4)
images_and_labels = list(zip(digits.images, digits.target))
for ax, (image, label) in zip(axes[0, :], images_and_labels[:4]):
    ax.set_axis_off()
    ax.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    ax.set_title('Training: %i' % label)

# To apply a classifier on this data, we need to flatten the image, to
# turn the data in a (samples, feature) matrix:
n_samples = len(digits.images)
data = digits.images.reshape((n_samples, -1))

# Create a classifier: a support vector classifier
classifier = svm.SVC(gamma=0.001)

# Split data into train and test subsets
X_train, X_test, y_train, y_test = train_test_split(
    data, digits.target, test_size=0.5, shuffle=False)

# We learn the digits on the first half of the digits
classifier.fit(X_train, y_train)

# Now predict the value of the digit on the second half:
predicted = classifier.predict(X_test)

images_and_predictions = list(zip(digits.images[n_samples // 2:], predicted))
for ax, (image, prediction) in zip(axes[1, :], images_and_predictions[:4]):
    ax.set_axis_off()
    ax.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    ax.set_title('Prediction: %i' % prediction)

print("Classification report for classifier %s:\n%s\n"
      % (classifier, metrics.classification_report(y_test, predicted)))
disp = metrics.plot_confusion_matrix(classifier, X_test, y_test)
disp.figure_.suptitle("Confusion Matrix")
print("Confusion matrix:\n%s" % disp.confusion_matrix)

plt.show()
```

▼ Scikit-image

Реализовывать различные операции для работы с изображениями можно как самостоятельно, работая с массивами numpy, так и используя специализированные библиотеки, например, scikit-image (<https://scikit-image.org/>). Ниже приведен пример использования Canny edge detector.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage as ndi

from skimage import feature

# Generate noisy image of a square
im = np.zeros((128, 128))
im[32:-32, 32:-32] = 1

im = ndi.rotate(im, 15, mode='constant')
im = ndi.gaussian_filter(im, 4)
im += 0.2 * np.random.random(im.shape)

# Compute the Canny filter for two values of sigma
edges1 = feature.canny(im)
edges2 = feature.canny(im, sigma=3)

# display results
fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3, figsize=(8, 3),
                                    sharex=True, sharey=True)

ax1.imshow(im, cmap=plt.cm.gray)
ax1.axis('off')
ax1.set_title('noisy image', fontsize=20)

ax2.imshow(edges1, cmap=plt.cm.gray)
ax2.axis('off')
ax2.set_title(r'Canny filter, $\sigma=1$', fontsize=20)

ax3.imshow(edges2, cmap=plt.cm.gray)
ax3.axis('off')
ax3.set_title(r'Canny filter, $\sigma=3$', fontsize=20)

fig.tight_layout()

plt.show()
```

▼ Tensorflow 2

Для создания и обучения нейросетевых моделей можно использовать фреймворк глубокого обучения Tensorflow 2. Ниже приведен пример простейшей нейронной сети, использующейся для классификации изображений из набора данных MNIST.

```
# Install TensorFlow

import tensorflow as tf

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation='softmax')
])

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

model.fit(x_train, y_train, epochs=5)

model.evaluate(x_test, y_test, verbose=2)
```

Для эффективной работы с моделями глубокого обучения убедитесь в том, что в текущей среде Google Colab используется аппаратный ускоритель GPU или TPU. Для смены среды выберите "среда выполнения" -> "сменить среду выполнения".

Большое количество tutorиалов и примеров с кодом на Tensorflow 2 можно найти на официальном сайте <https://www.tensorflow.org/tutorials?hl=ru>.

Также, Вам может понадобиться написать собственный генератор данных для Tensorflow 2. Скорее всего он будет достаточно простым, и его легко можно будет реализовать, используя официальную документацию TensorFlow 2. Но, на всякий случай (если не удалось сразу разобраться или хочется вникнуть в тему более глубоко), можете посмотреть следующий отличный tutorial: <https://stanford.edu/~shervine/blog/keras-how-to-generate-data-on-the-fly>.

Numba

В некоторых ситуациях, при ручных реализациях графовых алгоритмов, выполнение многократных вложенных циклов for в python можно существенно ускорить, используя JIT-компилятор Numba (<https://numba.pydata.org/>). Примеры использования Numba в Google Colab можно найти тут:

1. https://colab.research.google.com/github/cbnet/maldives/blob/master/numba/numba_cuda.ipynb
2. https://colab.research.google.com/github/evaneschneider/parallel-programming/blob/master/COMPASS_gpu_intro.ipynb

Пожалуйста, если Вы решили использовать Numba для решения этого практического задания, еще раз подумайте, нужно ли это Вам, и есть ли возможность реализовать требуемую функциональность иным способом. Используйте Numba только при реальной необходимости.

▼ Работа с zip архивами в Google Drive

Запаковка и распаковка zip архивов может пригодиться при сохранении и загрузки Вашей модели. Ниже приведен фрагмент кода, иллюстрирующий помещение нескольких файлов в zip архив с последующим чтением файлов из него. Все действия с директориями, файлами и архивами должны осуществляться с примонтированным Google Drive.

Создадим 2 изображения, поместим их в директорию tmp внутри PROJECT_DIR, запакуем директорию tmp в архив tmp.zip.

```
PROJECT_DIR = "/dev/prak_nn_1/"
arr1 = np.random.rand(100, 100, 3) * 255
arr2 = np.random.rand(100, 100, 3) * 255

img1 = Image.fromarray(arr1.astype('uint8'))
img2 = Image.fromarray(arr2.astype('uint8'))

p = "/content/drive/MyDrive/" + PROJECT_DIR

if not (Path(p) / 'tmp').exists():
    (Path(p) / 'tmp').mkdir()

img1.save(str(Path(p) / 'tmp' / 'img1.png'))
img2.save(str(Path(p) / 'tmp' / 'img2.png'))

%cd $p
!zip -r "tmp.zip" "tmp"
```

Распакуем архив tmp.zip в директорию tmp2 в PROJECT_DIR. Теперь внутри директории tmp2 содержится директория tmp, внутри которой находятся 2 изображения.

```
p = "/content/drive/MyDrive/" + PROJECT_DIR
%cd $p
!unzip -uq "tmp.zip" -d "tmp2"
```