#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 (5.2.0)
Requirement already satisfied: beautifulsoup4 in
/usr/local/lib/python3.10/dist-packages (from gdown) (4.12.3)
Requirement already satisfied: filelock in
/usr/local/lib/python3.10/dist-packages (from gdown) (3.16.1)
Requirement already satisfied: requests[socks] in
/usr/local/lib/python3.10/dist-packages (from gdown) (2.32.3)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-
packages (from gdown) (4.66.6)
Requirement already satisfied: soupsieve>1.2 in
/usr/local/lib/python3.10/dist-packages (from beautifulsoup4->gdown)
(2.6)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests[socks]->gdown)
(3.4.0)
Requirement already satisfied: idna<4,>=2.5 in
/usr/local/lib/python3.10/dist-packages (from requests[socks]->gdown)
(3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests[socks]->gdown)
(2.2.3)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests[socks]->gdown)
(2024.8.30)
Requirement already satisfied: PySocks!=1.5.7,>=1.5.6 in
/usr/local/lib/python3.10/dist-packages (from requests[socks]->gdown)
(1.7.1)
```

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 = True
TEST_ON_LARGE_DATASET = True
TISSUE_CLASSES = ('ADI', 'BACK', 'DEB', 'LYM', 'MUC', 'MUS', 'NORM',
'STR', 'TUM')
DATASETS_LINKS = { #собственные ссылки на наборы, т.к. советывали так
сделать
    'train': "1T5z6OPLoYUANn-_B14PSngZFm0mIOgsU",
    'train_small': "18v_U5xFYI3V9lIeewZ8WEUcX5Gwbn0hH",
    'train_tiny': "1CD-hIDZ8eWjSC4lz7Z6jy61OSUczujkC",
    'test': "1yrhk65_BzfPHDQpGPCZN4h8qwIeEmil2",
    'test_small': "1E2oMNP7YeS0Bs-xvRugAgYdEp-TvuDaW",
    'test_tiny': "1F0ve7Cl7c7Ln-7hoQLbFauaAjsVAh1g7"
}
```

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
import gdown
import time
import torch
from torch import nn
from torch.nn import functional as F
import torchvision
import torchvision.models as models
from torchvision.models import ResNet18_Weights
from torchvision.transforms import v2
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns
import warnings; warnings.filterwarnings(action='once')
from sklearn.metrics import confusion_matrix
from torch.optim.lr_scheduler import ReduceLROnPlateau
```

Класс Dataset

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

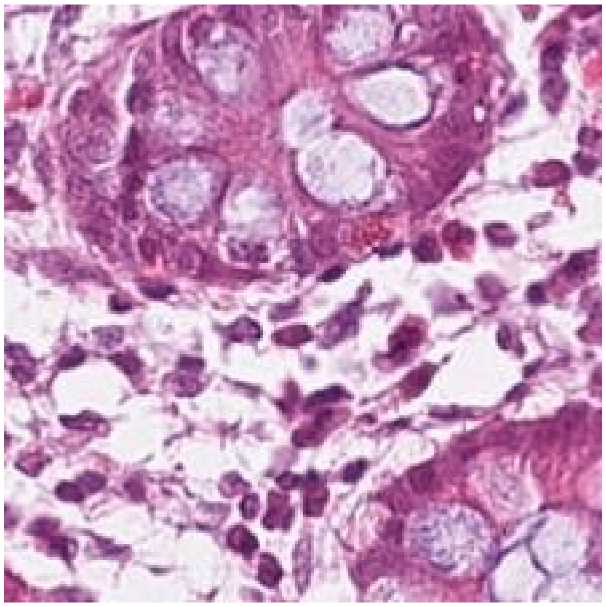
```
class Dataset:

    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]
```

Пример использвания 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.

```python
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[lbl]} class.')

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

```
<frozen importlib._bootstrap>:914: ImportWarning:
_PyDrive2ImportHook.find_spec() not found; falling back to
find_module()
<frozen importlib._bootstrap>:914: ImportWarning:
_PyDriveImportHook.find_spec() not found; falling back to
find_module()
<frozen importlib._bootstrap>:914: ImportWarning:
_GenerativeAIImportHook.find_spec() not found; falling back to
find_module()
<frozen importlib._bootstrap>:914: ImportWarning:
_OpenCVImportHook.find_spec() not found; falling back to find_module()
<frozen importlib._bootstrap>:914: ImportWarning:
APICoreClientInfoImportHook.find_spec() not found; falling back to
find_module()
<frozen importlib._bootstrap>:914: ImportWarning:
_BokehImportHook.find_spec() not found; falling back to find_module()
<frozen importlib._bootstrap>:914: ImportWarning:
_AltairImportHook.find_spec() not found; falling back to find_module()
Downloading...
From: https://drive.google.com/uc?export=download&confirm=pbef&id=1CD-
hIDZ8eWjSC4lz7Z6jy61OSUczujkC
To: /content/train_tiny.npz
100%|██████████| 105M/105M [00:00<00:00, 110MB/s]

Loading dataset train_tiny from npz.
Done. Dataset train_tiny consists of 900 images.

Got numpy array of shape (224, 224, 3), and label with code 6.
Label code corresponds to NORM class.
```



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 заргрузки 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 выбрки 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.

#Функция **k_folds**

Проверка входных данных: Проверяет, чтобы количество фолдов было хотя бы 2, и чтобы общее количество объектов было больше или равно количеству фолдов

Определение размера фолдов: Равномерно делит количество объектов на фолды Если объекты не делятся нацело на количество фолдов, остаток добавляется к первичным фолдам

Создание индексов для фолдов: Генерирует массив индексов объектов Делит объекты на фолды для каждого фолда создаются два массива Обучающие индексы (train_indices) Все объекты, за исключением объектов текущего фолда Тестовые индексы (test_indices) Объекты текущего фолда

Возврат: Возвращает список кортежей, где каждый кортеж содержит два массива: Обучающие индексы, Тестовые индексы.

Пример разбиения для 10 объектов и 3 фолдов: Размеры фолдов [4, 3, 3] Индексы объектов [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Разбиение: Фолд 1: обучающие индексы [4, 5, 6, 7, 8, 9], тестовые индексы [0, 1, 2, 3] Фолд 2: обучающие индексы [0, 1, 2, 3, 7, 8, 9], тестовые индексы [4, 5, 6] Фолд 3: обучающие индексы [0, 1, 2, 3, 4, 5, 6], тестовые индексы [7, 8, 9]

```
#функция для разбиения данных на фолды
def k_folds(num_objects: int, num_folds: int) ->
list[tuple[np.ndarray, np.ndarray]]:
    if num_folds < 2:
        raise ValueError("Split count must be at least 2")
    if num_objects < num_folds:
        raise ValueError("Total items must be greater than or equal to
the number of splits")

    fold_sizes = [num_objects // num_folds] * num_folds
    for i in range(num_objects % num_folds):
        fold_sizes[i] += 1

    indices = np.arange(num_objects, dtype=np.int32)
    splits = []
    start = 0
    for fold_size in fold_sizes:
        test_indices = indices[start:start + fold_size]
        train_indices = np.concatenate((indices[:start], indices[start
+ fold_size:]))
        splits.append((train_indices, test_indices))
        start += fold_size

    return splits

#определение архитектуры модели ResNet18 для классификации изображений
def create_model(base_model, num_ftrs, num_classes):
    for param in base_model.parameters():
        param.requires_grad = False

    num_ftrs = base_model.fc.in_features
    base_model.fc = torch.nn.Linear(num_ftrs, num_classes)
    return base_model
```

класс **Dataloader**

Аргументы конструктора: dataset: Датасет, который предоставляет изображения и метки

indices: Индексы объектов в датасете, которые будут использованы для создания выборки

batch_size: Размер пакета(батча), который будет извлечён из выборки за одну итерацию

transforms: Преобразования, которые будут применяться к изображениям

augmenting_transforms: Аугментации для увеличения разнообразия данных

limit: Процент данных, который нужно использовать(от 0 до 1)

Итератор: iter: Метод, который возвращает сам объект и инициализирует итерацию, перемешивая индексы данных для каждой новой итерации

next: Метод, который извлекает следующий батч данных, используя индексы Каждое изображение: Загружается из датасета Применяются аугментации и трансформации, если они заданы

Обработка батчей: Тензоры batch_images и batch_labels заполняются для каждого индекса в текущем батче

Преобразования и аугментации могут применяться в любом порядке, в зависимости от того, как они передаются в конструктор

Сохранение изображений в нужном формате: Изображения преобразуются в тензоры с размерностью (batch_size, 3, 224, 224) что соответствует формату, используемому в большинстве CNN

Перемешивание индексов: В iter индексы перемешиваются перед каждой итерацией, что помогает улучшить обобщающую способность модели за счет случайности в выборке данных

```python
class Dataloader:
    def __init__(self, dataset: Dataset, indices: np.ndarray,
batch_size: int, transforms=None, augmenting_transforms=None,
limit=1.0):
        if not (0 < limit <= 1.0):
            raise ValueError("Data limit must be between 0 and 1")

        self.dataset = dataset
        self.indices = indices[:int(len(indices) * limit)]
        self.batch_size = batch_size
        self.transforms = transforms
        self.augmenting_transforms = augmenting_transforms
        self.num_batches = len(self.indices) // batch_size
        self.current_batch_idx = 0

    def __iter__(self):
        self.current_batch_idx = 0
        np.random.shuffle(self.indices)
        return self

    def __next__(self):
        if self.current_batch_idx < self.num_batches:
            batch_start = self.current_batch_idx * self.batch_size
            batch_end = batch_start + self.batch_size
            selected_indices = self.indices[batch_start:batch_end]

            batch_images = torch.empty((self.batch_size, 3, 224, 224),
dtype=torch.float32)
            batch_labels = torch.empty(self.batch_size,
dtype=torch.long)

            for idx, data_idx in enumerate(selected_indices):
                image, label = self.dataset.image_with_label(data_idx)
                image = torch.from_numpy(image).permute(2, 0, 1)

                if self.augmenting_transforms:
                    image = self.augmenting_transforms(image)
                if self.transforms:
                    image = self.transforms(image)

                batch_images[idx] = image
                batch_labels[idx] = torch.tensor(label)

            self.current_batch_idx += 1
            return batch_images, batch_labels
        else:
            raise StopIteration
```

класс **CrossValidator**

Аргументы конструктора init: dataset: набор данных, который будет разделён на фолды

batch_size: размер батча для обучения и валидации

num_folds: число фолдов для кросс-валидации. Должно быть не менее 2

transforms: преобразования, применяемые к данным в процессе обучения

augmenting_transforms: аугментации для увеличения разнообразия данных

seed: фиксация случайности для воспроизводимости

limit: ограничения размера данных для обучающей и валидационной выборов(значения в диапазоне (0, 1])

```
#LBL2
class CrossValidator:
    def __init__(self, dataset: Dataset, batch_size: int, num_folds:
int, transforms=None, augmenting_transforms=None, seed=42, limit=(1.0,
1.0)):
        if num_folds < 2:
            raise ValueError("Number of folds must be at least 2")
        if len(limit) != 2 or not all(0 < value <= 1 for value in
limit):
            raise ValueError("Data limits must be within the range 0
and 1")

        self.dataset = dataset
        self.batch_size = batch_size
        self.num_folds = num_folds
        self.transforms = transforms
        self.augmenting_transforms = augmenting_transforms
        self.limit = limit

        self.indices = np.arange(dataset.n_files)
        np.random.seed(seed)
        np.random.shuffle(self.indices)

        self.fold_indices = k_folds(len(self.indices), num_folds)
        self.current_fold = 0

    def get_train_val(self):
        train_idx = self.indices[self.fold_indices[self.current_fold]
[0]]
        val_idx = self.indices[self.fold_indices[self.current_fold]
[1]]

        self.current_fold = (self.current_fold + 1) % self.num_folds

        return (
            Dataloader(self.dataset, train_idx, self.batch_size,
                       self.transforms, self.augmenting_transforms,
limit=self.limit[0]),
            Dataloader(self.dataset, val_idx, self.batch_size,
                       self.transforms, self.augmenting_transforms,
limit=self.limit[1])
        )

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

#LBL8 #LBL6 #LBL4
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()
```

Матрица Ошибок

матрицы ошибок, которая показывает, как предсказания модели соотносятся с истинными метками классов. Функция также вычисляет True Positive (TP) и False Positive (FP) для каждого класса.

Используем confusion_matrix из библиотеки sklearn для подсчёта количества совпадений между предсказанными и истинными метками классов.

Визуализируем матрицу ошибок: Построение тепловой карты (heatmap) с использованием библиотеки seaborn для наглядного отображения.

Вычисляет метрики для каждого класса: TP (True Positive): Количество правильных предсказаний для данного класса. FP (False Positive): Количество случаев, когда данный класс был предсказан неправильно. Выводит метрики в табличной форме.

```
def confi_matrix(gt: List[int], pred: List[int], class_names: 
List[str]):
    matrix = confusion_matrix(gt, pred)

    plt.figure(figsize=(10, 8), dpi=100)
    sns.heatmap(matrix,
                xticklabels=class_names,
                yticklabels=class_names,
                cmap='RdYlGn',
                center=0,
                annot=True,
                fmt="d",
                cbar_kws={'label': 'Number of Samples'})

    plt.title('Confusion Matrix', fontsize=22, pad=20)
    plt.xlabel('Predicted Labels', fontsize=16, labelpad=10)
    plt.ylabel('True Labels', fontsize=16, labelpad=10)
    plt.xticks(fontsize=12, rotation=45)
    plt.yticks(fontsize=12)
    plt.show()
    TP = np.diag(matrix)
    FP = matrix.sum(axis=0) - TP

    print(f"{'Class':<15}{'True Positive (TP)':<20}{'False Positive 
(FP)':<20}")
    print("-" * 55)
    for i, class_name in enumerate(class_names):
        print(f"{class_name:<15}{TP[i]:<20}{FP[i]:<20}")
```

Почему так

Изначально я пытался сделать с помощью классического МЛ'ля, но перенабивание ОЗУ я не осилил. По советам уже знающих людей, решил сделать именно свертку, да и примеров на том же Гитхабе и кагле было достаточно, что бы человек в первые который с этим знакомится мог что-то и написать, ну, я брал идеи и части кода других, но в основном все делалось с моей руки и могу даже защитить данную работу.

Я использую предобученную модель ResNet18 и дообучаю ее на новом наборе данных, чтобы она научилась классифицировать изображения на заданных классах. Даже попытался предотвратить переобучение, но, не знаю, наверное надо было брать меньше эпох.

На чем тестировал: На полном тесте: accuracy 0.9847, balanced accuracy 0.9847

На маленьком тесте: accuracy 0.9667, balanced accuracy 0.9667

```
class Model:
    def __init__(self):
        self.model = 
create_model(torchvision.models.resnet18(pretrained=True), 6, 
9).to(device)
        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/DP/{name}.pth')

    def load(self, name: str):
        name_to_id_dict = {
            "best": '1-KG_C4OaeYnh9DpB4GH0_fdUoLK706Fg'
        }
        output = f"{name}.pth"
        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()
    #LBL1
    def evaluate(self, dataloader: Dataloader, loss_fn, weights):
        losses = []
        num_correct = np.zeros(9, dtype=int)
        num_elements = np.zeros(9, dtype=int)

        for batch in dataloader:
            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] ==
np.arange(9)).sum(axis=0)
                num_correct += ((y_pred.numpy()[:, None] ==
np.arange(9)) & (y_pred.numpy()[:, None] == y_batch.numpy()[:,
None])).sum(axis=0)

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

        accuracy = num_correct.sum() / num_elements.sum()
        return accuracy, np.mean(losses), weights

    def train(self, dataset: Dataset):
        optim = torch.optim.AdamW
        lossF = torch.nn.CrossEntropyLoss
        learning_rate = 1e-4
        weight_decay = 1e-4

        #LBL9
        optimizer = optim(self.model.parameters(), lr=learning_rate,
weight_decay=weight_decay)
        scheduler =
torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, "min",
patience=5, threshold=0.01, threshold_mode="rel")

        augmenting_transforms = v2.Compose([
            v2.RandomHorizontalFlip(),
            v2.RandomVerticalFlip(),
            v2.RandomRotation(degrees=10),
            v2.RandomResizedCrop(size=(224, 224), scale=(0.8, 1.0))])

        batch_size = 32
        folds = 10
        limit = (0.3, 1.0)
        dataloader = CrossValidator(dataset, batch_size, folds,
self.transforms, augmenting_transforms, limit=limit)

        history = ""
        train_loss_history, train_acc_history = [], []
        val_loss_history, val_acc_history = [], []

        local_train_loss_history, local_train_acc_history = [], []

        n_epoch = 60
        eval_every = 30

        weights = torch.ones(9).to(device)

        #LBL3
        for epoch in range(n_epoch):
            print(f"Epoch {epoch+1}/{n_epoch}")
            history += f"Epoch {epoch+1}\n"
            loss_fn =
lossF(weight=weights.type(torch.FloatTensor).to(device))

            train, val = dataloader.get_train_val()

            self.model.train()

            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:
                    history += f"avg train loss {eval_every}
iterations: {np.mean(local_train_loss_history)} accuracy:
{np.mean(local_train_acc_history)} \n"
                    print(f"avg train loss {eval_every} iterations:
{np.mean(local_train_loss_history)} accuracy:
{np.mean(local_train_acc_history)}")
            self.model.eval()
            val_accuracy, val_loss, weights = self.evaluate(val,
loss_fn, weights)
            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)
            plot_train_process(train_loss_history, val_loss_history,
train_acc_history, val_acc_history)

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

    def test_on_dataset(self, dataset: Dataset, limit=None):
        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):
        img_tensor = self.transforms(torch.from_numpy(img).permute(2,
0, 1)).unsqueeze(0).to(device)
        prediction = self.model(img_tensor)
        return int(torch.argmax(prediction, dim=1).cpu()[0])
```

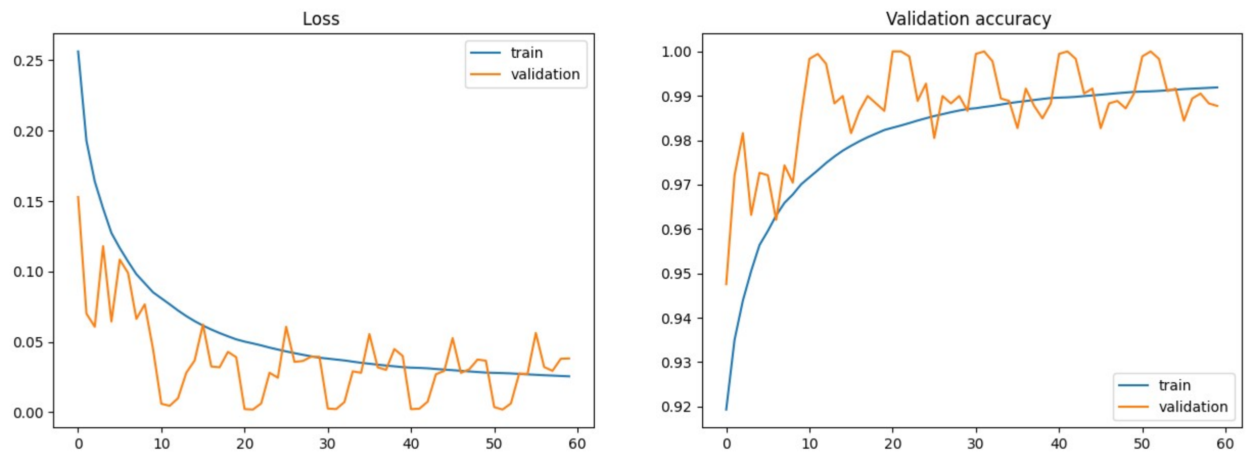
Классификация изображений

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")
d_test = Dataset("test")

Downloading...
From: https://drive.google.com/uc?
```

```
export=download&confirm=pbef&id=1T5z6OPLoYUANn-_B14PSngZFm0mIOgsU
To: /content/train.npz
100%|██████████| 2.10G/2.10G [00:23<00:00, 88.7MB/s]

Loading dataset train from npz.
Done. Dataset train consists of 18000 images.

Downloading...
From: https://drive.google.com/uc?
export=download&confirm=pbef&id=1yrhk65_BzfPHDQpGPCZN4h8qwIeEmil2
To: /content/test.npz
100%|██████████| 525M/525M [00:04<00:00, 124MB/s]

Loading dataset test from npz.
Done. Dataset test consists of 4500 images.
```

```python
model = Model()

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

model.train(d_train)
model.save("best")
```



```
Epoch 1
avg train loss 30 iterations: 0.3265902534127235 accuracy:
0.893750011920929
avg train loss 30 iterations: 0.31767323339978853 accuracy:
0.9005208611488342
avg train loss 30 iterations: 0.29132012592421636 accuracy:
0.9083333611488342
```

```
avg train loss 30 iterations: 0.2784990563367804 accuracy:
0.9117187261581421
avg train loss 30 iterations: 0.2569602446382244 accuracy:
0.9191666841506958
Epoch 1/60: val loss: 0.1528935681895486 accuracy: 0.9475446428571429
New weights tensor([0.9004, 1.0000, 0.4247, 0.9724, 0.7038, 0.8030,
0.9120, 0.3791, 0.7032])
Learning rate 0.0001
Epoch 2
avg train loss 30 iterations: 0.23684837002286596 accuracy:
0.924723744392395
avg train loss 30 iterations: 0.22731804558168656 accuracy:
0.9265402555465698
avg train loss 30 iterations: 0.21132242223982123 accuracy:
0.9306275844573975
avg train loss 30 iterations: 0.20040630466026355 accuracy:
0.933002769947052
avg train loss 30 iterations: 0.19260817749481463 accuracy:
0.9350082874298096
Epoch 2/60: val loss: 0.06999929747377921 accuracy: 0.9720982142857143

New weights tensor([1.0000, 1.0000, 0.7542, 0.8597, 0.9403, 0.9129,
0.8880, 0.6702, 0.6105])
Learning rate 0.0001
Epoch 3
avg train loss 30 iterations: 0.18454415875442146 accuracy:
0.9376882314682007
avg train loss 30 iterations: 0.17662811427613853 accuracy:
0.940348744392395
avg train loss 30 iterations: 0.17281110587468065 accuracy: 0.94140625

avg train loss 30 iterations: 0.16906937908588673 accuracy:
0.9423133730888367
avg train loss 30 iterations: 0.16413018414318 accuracy:
0.9437915086746216
Epoch 3/60: val loss: 0.06066995052554246 accuracy: 0.9815848214285714

New weights tensor([0.9702, 1.0000, 0.8842, 0.9129, 0.9199, 0.6443,
0.8367, 0.9720, 0.9412])
Learning rate 0.0001
Epoch 4
avg train loss 30 iterations: 0.1587704548537037 accuracy:
0.9453933835029602
avg train loss 30 iterations: 0.15443484159368026 accuracy:
0.9469420313835144
avg train loss 30 iterations: 0.15146758355471282 accuracy:
0.9478591084480286
avg train loss 30 iterations: 0.14817551768630302 accuracy:
0.9495527744293213
avg train loss 30 iterations: 0.14447856707047443 accuracy:
```

```
0.9506633281707764
Epoch 4/60: val loss: 0.11794914906412098 accuracy: 0.9631696428571429

New weights tensor([0.9701, 0.9698, 0.7778, 0.9385, 0.8629, 0.9080,
0.7882, 0.3714, 0.8905])
Learning rate 0.0001
Epoch 5
avg train loss 30 iterations: 0.14034303072884044 accuracy:
0.9518434405326843
avg train loss 30 iterations: 0.136237700512571 accuracy: 0.953125
avg train loss 30 iterations: 0.13336930174839493 accuracy:
0.9541606903076172
avg train loss 30 iterations: 0.13040718713579752 accuracy:
0.95524001121521
avg train loss 30 iterations: 0.1274312095017486 accuracy:
0.9563162922859192
Epoch 5/60: val loss: 0.064429230514049 accuracy: 0.97265625
New weights tensor([0.9695, 1.0000, 0.9437, 0.9098, 0.9701, 0.8543,
0.9420, 0.4857, 0.6639])
Learning rate 0.0001
Epoch 6
avg train loss 30 iterations: 0.12503400536426076 accuracy:
0.9570859670639038
avg train loss 30 iterations: 0.12174338187702412 accuracy:
0.9581671953201294
avg train loss 30 iterations: 0.11973875655335259 accuracy:
0.9586908221244812
avg train loss 30 iterations: 0.11767784655732767 accuracy:
0.9591428637504578
avg train loss 30 iterations: 0.11667699678466548 accuracy:
0.9595304131507874
Epoch 6/60: val loss: 0.10839391321600747 accuracy: 0.9720982142857143

New weights tensor([1.0000, 0.9702, 0.6702, 0.9664, 0.8672, 0.9445,
0.6992, 0.6717, 0.8745])
Learning rate 0.0001
Epoch 7
avg train loss 30 iterations: 0.11433793157169946 accuracy:
0.960403323173523
avg train loss 30 iterations: 0.11228209750714285 accuracy:
0.9611477851867676
avg train loss 30 iterations: 0.11038330346827459 accuracy:
0.9620042443275452
avg train loss 30 iterations: 0.10887341405656262 accuracy:
0.9624756574630737
avg train loss 30 iterations: 0.10706420034760165 accuracy:
0.9630385637283325
Epoch 7/60: val loss: 0.09889261914317363 accuracy: 0.9620535714285714

New weights tensor([1.0000, 1.0000, 0.8306, 0.9712, 0.7171, 0.8783,
0.7875, 0.3691, 0.7767])
Learning rate 0.0001
Epoch 8
avg train loss 30 iterations: 0.10511450399601834 accuracy:
0.9636039733886719
avg train loss 30 iterations: 0.10325244527131706 accuracy:
0.9642737507820129
avg train loss 30 iterations: 0.10152957942876753 accuracy:
0.9648812413215637
avg train loss 30 iterations: 0.09984158400838646 accuracy:
0.9652187824249268
avg train loss 30 iterations: 0.09810593285402645 accuracy:
0.9658502340316772
Epoch 8/60: val loss: 0.0662521438145112 accuracy: 0.9743303571428571
New weights tensor([1.0000, 1.0000, 0.8557, 1.0000, 0.9654, 0.8497,
1.0000, 0.5388, 0.6353])
Learning rate 0.0001
Epoch 9
avg train loss 30 iterations: 0.09663539184806552 accuracy:
0.966251015663147
avg train loss 30 iterations: 0.09530922858527793 accuracy:
0.9666551947593689
avg train loss 30 iterations: 0.09426472361023051 accuracy:
0.9669684171676636
avg train loss 30 iterations: 0.09265563665473335 accuracy:
0.9674792885780334
avg train loss 30 iterations: 0.09164092824693196 accuracy:
0.9677374958992004
Epoch 9/60: val loss: 0.07662267701899898 accuracy: 0.9704241071428571

New weights tensor([0.8847, 1.0000, 0.9051, 1.0000, 0.8169, 0.8425,
0.9699, 0.5745, 0.6425])
Learning rate 1e-05
Epoch 10
avg train loss 30 iterations: 0.09042816899780216 accuracy:
0.9681875705718994
avg train loss 30 iterations: 0.08915432115304307 accuracy:
0.968661904335022
avg train loss 30 iterations: 0.08761306364049723 accuracy:
0.9691813588142395
avg train loss 30 iterations: 0.08639712550707584 accuracy:
0.9696162939071655
avg train loss 30 iterations: 0.08521507076391083 accuracy:
0.9700339436531067
Epoch 10/60: val loss: 0.04486510725733491 accuracy:
0.9854910714285714
New weights tensor([1.0000, 0.9385, 0.8858, 1.0000, 1.0000, 0.8847,
0.9719, 0.7138, 0.8666])
Learning rate 1e-05
```

```
Epoch 11
avg train loss 30 iterations: 0.08462615796559878 accuracy:
0.9702922105789185
avg train loss 30 iterations: 0.0836938544099403 accuracy:
0.9706408977508545
avg train loss 30 iterations: 0.08282533496916585 accuracy:
0.9709374904632568
avg train loss 30 iterations: 0.08194457316889707 accuracy:
0.9712806940078735
avg train loss 30 iterations: 0.08092621281940815 accuracy:
0.9716491103172302
Epoch 11/60: val loss: 0.005999669882450169 accuracy:
0.9983258928571429
New weights tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.9391,
1.0000, 1.0000, 0.9693])
Learning rate 1e-05
Epoch 12
avg train loss 30 iterations: 0.0800875094836266 accuracy:
0.9719840288162231
avg train loss 30 iterations: 0.07909392714373256 accuracy:
0.9723634719848633
avg train loss 30 iterations: 0.07839996948346777 accuracy:
0.9726406335830688
avg train loss 30 iterations: 0.0776782326120125 accuracy:
0.972838282585144
avg train loss 30 iterations: 0.07670097362281203 accuracy:
0.9732019305229187
Epoch 12/60: val loss: 0.004500384974692549 accuracy:
0.9994419642857143
New weights tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
1.0000, 1.0000, 0.9688])
Learning rate 1e-05
Epoch 13
avg train loss 30 iterations: 0.07573437227715357 accuracy:
0.9735511541366577
avg train loss 30 iterations: 0.07490533326740544 accuracy:
0.9738581776618958
avg train loss 30 iterations: 0.07405680464660223 accuracy:
0.9741719365119934
avg train loss 30 iterations: 0.07314844594149528 accuracy:
0.9745244383811951
avg train loss 30 iterations: 0.07228200259062136 accuracy:
0.9748502969741821
Epoch 13/60: val loss: 0.009929785473754496 accuracy:
0.9972098214285714
New weights tensor([1.0000, 1.0000, 0.9406, 1.0000, 1.0000, 1.0000,
0.9712, 0.9720, 0.9704])
Learning rate 1e-05
Epoch 14
```

```
avg train loss 30 iterations: 0.07146028578022053 accuracy:
0.9751316905021667
avg train loss 30 iterations: 0.07062763313492838 accuracy:
0.9754541516304016
avg train loss 30 iterations: 0.06984670403677946 accuracy:
0.9757062792778015
avg train loss 30 iterations: 0.06911684715340385 accuracy:
0.9759811758995056
avg train loss 30 iterations: 0.06826965267987287 accuracy:
0.9763073921203613
Epoch 14/60: val loss: 0.028033147172079356 accuracy: 0.98828125
New weights tensor([1.0000, 1.0000, 0.9695, 0.9687, 0.8890, 0.8223,
0.9041, 0.9492, 0.8910])
Learning rate 1e-05
Epoch 15
avg train loss 30 iterations: 0.06745889273536962 accuracy:
0.9766062498092651
avg train loss 30 iterations: 0.06666410400786757 accuracy:
0.9769290685653687
avg train loss 30 iterations: 0.06595009732552523 accuracy:
0.9772005677223206
avg train loss 30 iterations: 0.0652881598104704 accuracy:
0.9774227738380432
avg train loss 30 iterations: 0.06469831859842358 accuracy:
0.97762531042099
Epoch 15/60: val loss: 0.03669829175071625 accuracy:
0.9899553571428571
New weights tensor([1.0000, 1.0000, 0.8900, 1.0000, 0.9406, 0.9107,
0.9707, 0.8543, 0.9116])
Learning rate 1e-05
Epoch 16
avg train loss 30 iterations: 0.06395258928391949 accuracy:
0.9779003262519836
avg train loss 30 iterations: 0.06332579202772248 accuracy:
0.9781183004379272
avg train loss 30 iterations: 0.06278052028382766 accuracy:
0.9782776236534119
avg train loss 30 iterations: 0.062131084088969245 accuracy:
0.9785246253013611
avg train loss 30 iterations: 0.061532361014369166 accuracy:
0.9787266850471497
Epoch 16/60: val loss: 0.06231858510935646 accuracy:
0.9815848214285714
New weights tensor([1.0000, 0.9701, 0.8944, 1.0000, 0.9180, 0.8648,
0.7767, 0.7714, 0.8739])
Learning rate 1e-05
Epoch 17
avg train loss 30 iterations: 0.06091896481661127 accuracy:
0.9789451956748962
```

```
avg train loss 30 iterations: 0.06032900575087921 accuracy:
0.9791498184204102
avg train loss 30 iterations: 0.059812406387345654 accuracy:
0.9793495535850525
avg train loss 30 iterations: 0.059270585117179385 accuracy:
0.9795322418212891
avg train loss 30 iterations: 0.05875280023472312 accuracy:
0.9797471761703491
Epoch 17/60: val loss: 0.03245937498702135 accuracy:
0.9866071428571429
New weights tensor([1.0000, 1.0000, 0.9400, 1.0000, 0.9426, 0.7974,
0.8885, 0.8040, 0.9397])
Learning rate 1e-05
Epoch 18
avg train loss 30 iterations: 0.058235822687236004 accuracy:
0.9799407720565796
avg train loss 30 iterations: 0.05772720539236137 accuracy:
0.9801222681999207
avg train loss 30 iterations: 0.05717684139646241 accuracy:
0.9803231954574585
avg train loss 30 iterations: 0.05672294753566161 accuracy:
0.9804731011390686
avg train loss 30 iterations: 0.05622287881802923 accuracy:
0.9806657433509827
Epoch 18/60: val loss: 0.03188250181067685 accuracy:
0.9899553571428571
New weights tensor([1.0000, 1.0000, 0.9112, 1.0000, 0.8992, 0.9478,
0.8847, 0.8836, 0.9431])
Learning rate 1.0000000000000002e-06
Epoch 19
avg train loss 30 iterations: 0.05572712357552968 accuracy:
0.9808383584022522
avg train loss 30 iterations: 0.05530592554550049 accuracy:
0.9809890389442444
avg train loss 30 iterations: 0.054822277250521675 accuracy:
0.9811587333679199
avg train loss 30 iterations: 0.054368120446464246 accuracy:
0.9813358783721924
avg train loss 30 iterations: 0.05392894589208754 accuracy:
0.9814766645431519
Epoch 19/60: val loss: 0.04286966848979189 accuracy: 0.98828125
New weights tensor([1.0000, 1.0000, 0.9051, 0.9732, 0.9348, 0.8930,
0.9409, 0.8629, 0.8910])
Learning rate 1.0000000000000002e-06
Epoch 20
avg train loss 30 iterations: 0.053490499499452386 accuracy:
0.9816315770149231
avg train loss 30 iterations: 0.0530299652046307 accuracy:
0.9817984104156494
```

```
avg train loss 30 iterations: 0.052635859545003875 accuracy:
0.9819512367248535
avg train loss 30 iterations: 0.05218097201554008 accuracy:
0.9821323752403259
avg train loss 30 iterations: 0.05174897515065788 accuracy:
0.9822892546653748
Epoch 20/60: val loss: 0.03904132094183816 accuracy:
0.9866071428571429
New weights tensor([1.0000, 0.9391, 0.9129, 1.0000, 0.9412, 0.8564,
0.8910, 0.7648, 1.0000])
Learning rate 1.0000000000000002e-06
Epoch 21
avg train loss 30 iterations: 0.05149309595339725 accuracy:
0.9823770523071289
avg train loss 30 iterations: 0.05123357384257939 accuracy:
0.9824573993682861
avg train loss 30 iterations: 0.05085857713497147 accuracy:
0.9825763702392578
avg train loss 30 iterations: 0.050546046508693765 accuracy:
0.9826831221580505
avg train loss 30 iterations: 0.05016878702338456 accuracy:
0.9828075766563416
Epoch 21/60: val loss: 0.002191469389539894 accuracy: 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.])
Learning rate 1.0000000000000002e-06
Epoch 22
avg train loss 30 iterations: 0.04986912216545309 accuracy:
0.9829252362251282
avg train loss 30 iterations: 0.049609722557334396 accuracy:
0.9830354452133179
avg train loss 30 iterations: 0.0493467547492033 accuracy:
0.9831435680389404
avg train loss 30 iterations: 0.0491515376139892 accuracy:
0.9832118153572083
avg train loss 30 iterations: 0.04886003200643623 accuracy:
0.9833070039749146
Epoch 22/60: val loss: 0.0018539890227527525 accuracy: 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.])
Learning rate 1.0000000000000002e-06
Epoch 23
avg train loss 30 iterations: 0.04858257437493613 accuracy:
0.9834147691726685
avg train loss 30 iterations: 0.048278962875960646 accuracy:
0.9835249185562134
avg train loss 30 iterations: 0.04802191572581215 accuracy:
0.9836422801017761
avg train loss 30 iterations: 0.04771146717442623 accuracy:
0.9837576150894165
avg train loss 30 iterations: 0.047455641754215135 accuracy:
```

```
0.9838529825210571
Epoch 23/60: val loss: 0.006241410445259784 accuracy:
0.9988839285714286
New weights tensor([1.0000, 1.0000, 0.9698, 1.0000, 1.0000, 1.0000,
1.0000, 1.0000, 0.9707])
Learning rate 1.0000000000000002e-06
Epoch 24
avg train loss 30 iterations: 0.04712828288053092 accuracy:
0.9839690923690796
avg train loss 30 iterations: 0.046798358358490694 accuracy:
0.9840875267982483
avg train loss 30 iterations: 0.04648604372325887 accuracy:
0.9842039942741394
avg train loss 30 iterations: 0.04619133439062729 accuracy:
0.9843097925186157
avg train loss 30 iterations: 0.04589697190057626 accuracy:
0.9844224452972412
Epoch 24/60: val loss: 0.0280255439470888 accuracy: 0.9888392857142857

New weights tensor([1.0000, 1.0000, 0.9107, 0.9687, 0.8890, 0.9080,
0.9030, 0.9252, 0.9169])
Learning rate 1.0000000000000002e-06
Epoch 25
avg train loss 30 iterations: 0.04557868405356913 accuracy:
0.9845374822616577
avg train loss 30 iterations: 0.04535255704526065 accuracy:
0.9846294522285461
avg train loss 30 iterations: 0.04505155924041413 accuracy:
0.9847368001937866
avg train loss 30 iterations: 0.04474920880301607 accuracy:
0.9848507642745972
avg train loss 30 iterations: 0.04451737142744726 accuracy:
0.9849380850791931
Epoch 25/60: val loss: 0.024470670015164484 accuracy:
0.9927455357142857
New weights tensor([1.0000, 1.0000, 0.9165, 1.0000, 0.9403, 0.9400,
1.0000, 0.9107, 0.9116])
Learning rate 1.0000000000000002e-06
Epoch 26
avg train loss 30 iterations: 0.044254204598382425 accuracy:
0.9850361347198486
avg train loss 30 iterations: 0.04396371029564325 accuracy:
0.9851369261741638
avg train loss 30 iterations: 0.043715467066454364 accuracy:
0.9852280020713806
avg train loss 30 iterations: 0.04342873224771739 accuracy:
0.9853417873382568
avg train loss 30 iterations: 0.04316293419871901 accuracy:
0.9854378700256348
```

```
Epoch 26/60: val loss: 0.06069443895830773 accuracy: 0.98046875
New weights tensor([1.0000, 0.9704, 0.8695, 1.0000, 0.8925, 0.8404,
0.7767, 0.7693, 0.9041])
Learning rate 1.0000000000000002e-06
Epoch 27
avg train loss 30 iterations: 0.0429163356862282 accuracy:
0.9855204224586487
avg train loss 30 iterations: 0.0426554884573704 accuracy:
0.9856137037277222
avg train loss 30 iterations: 0.042450906929716946 accuracy:
0.9856978058815002
avg train loss 30 iterations: 0.04219682235044871 accuracy:
0.9857884049415588
avg train loss 30 iterations: 0.04196522939196413 accuracy:
0.9858623743057251
Epoch 27/60: val loss: 0.03572838554516368 accuracy:
0.9899553571428571
New weights tensor([1.0000, 1.0000, 0.8842, 1.0000, 0.9423, 0.9382,
0.9429, 0.8030, 0.9693])
Learning rate 1.0000000000000002e-06
Epoch 28
avg train loss 30 iterations: 0.041686701533561896 accuracy:
0.985969066619873
avg train loss 30 iterations: 0.041462279982799964 accuracy:
0.98604816198349
avg train loss 30 iterations: 0.041230029186085475 accuracy:
0.9861261248588562
avg train loss 30 iterations: 0.04097376773559371 accuracy:
0.9862253069877625
avg train loss 30 iterations: 0.04077590978926097 accuracy:
0.9863008856773376
Epoch 28/60: val loss: 0.03636500919726261 accuracy: 0.98828125
New weights tensor([1.0000, 1.0000, 0.9112, 1.0000, 0.9656, 0.9219,
0.9120, 0.8040, 0.8890])
Learning rate 1.0000000000000002e-07
Epoch 29
avg train loss 30 iterations: 0.04054529197461728 accuracy:
0.9863932728767395
avg train loss 30 iterations: 0.040370508788565705 accuracy:
0.9864447116851807
avg train loss 30 iterations: 0.04013286755260419 accuracy:
0.986531674861908
avg train loss 30 iterations: 0.03990482948148553 accuracy:
0.986617386341095
avg train loss 30 iterations: 0.03969952053862008 accuracy:
0.9866877198219299
Epoch 29/60: val loss: 0.039398414123622515 accuracy:
0.9899553571428571
New weights tensor([0.9702, 1.0000, 0.9046, 1.0000, 0.9668, 0.8672,
0.9701, 0.8895, 0.9173])
Learning rate 1.0000000000000002e-07
Epoch 30
avg train loss 30 iterations: 0.039465041245270165 accuracy:
0.986774206161499
avg train loss 30 iterations: 0.03925001173545635 accuracy:
0.9868494868278503
avg train loss 30 iterations: 0.0391033670215741 accuracy:
0.9868888258934021
avg train loss 30 iterations: 0.038880711251573365 accuracy:
0.9869693517684937
avg train loss 30 iterations: 0.03869119742020792 accuracy:
0.9870280623435974
Epoch 30/60: val loss: 0.0393672070042937 accuracy: 0.9866071428571429

New weights tensor([1.0000, 0.9094, 0.9129, 1.0000, 0.9129, 0.9400,
0.9177, 0.7389, 0.9722])
Learning rate 1.0000000000000002e-07
Epoch 31
avg train loss 30 iterations: 0.03851799337187498 accuracy:
0.9870819449424744
avg train loss 30 iterations: 0.03841487664359313 accuracy:
0.9871119260787964
avg train loss 30 iterations: 0.03831143084685701 accuracy:
0.987127959728241
avg train loss 30 iterations: 0.03817974709079511 accuracy:
0.9871639609336853
avg train loss 30 iterations: 0.038074255507323955 accuracy:
0.9871995449066162
Epoch 31/60: val loss: 0.0025744971696570118 accuracy:
0.9994419642857143
New weights tensor([1.0000, 1.0000, 0.9711, 1.0000, 1.0000, 1.0000,
1.0000, 1.0000, 1.0000])
Learning rate 1.0000000000000002e-07
Epoch 32
avg train loss 30 iterations: 0.037929612934962824 accuracy:
0.9872439503669739
avg train loss 30 iterations: 0.037787308371279124 accuracy:
0.987298309803009
avg train loss 30 iterations: 0.037640105448440216 accuracy:
0.9873650670051575
avg train loss 30 iterations: 0.037539568116669295 accuracy:
0.9874244928359985
avg train loss 30 iterations: 0.03739674058859453 accuracy:
0.9874832034111023
Epoch 32/60: val loss: 0.002188035493548211 accuracy: 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.])
Learning rate 1.0000000000000002e-07
Epoch 33
avg train loss 30 iterations: 0.03725862917760391 accuracy:
```

```
0.9875244498252869
avg train loss 30 iterations: 0.03714076064072796 accuracy:
0.9875689744949341
avg train loss 30 iterations: 0.03706937859124174 accuracy:
0.9876003265380859
avg train loss 30 iterations: 0.03691672725332728 accuracy:
0.9876564741134644
avg train loss 30 iterations: 0.03676085097982626 accuracy:
0.9877245426177979
Epoch 33/60: val loss: 0.007170992033154887 accuracy:
0.9977678571428571
New weights tensor([1.0000, 1.0000, 0.9699, 1.0000, 0.9724, 1.0000,
1.0000, 0.9719, 0.9707])
Learning rate 1.0000000000000002e-07
Epoch 34
avg train loss 30 iterations: 0.03657243750959155 accuracy:
0.9877942204475403
avg train loss 30 iterations: 0.03639320426439745 accuracy:
0.9878606200218201
avg train loss 30 iterations: 0.03623517309158538 accuracy:
0.9879201054573059
avg train loss 30 iterations: 0.0360665472684655 accuracy:
0.9879727363586426
avg train loss 30 iterations: 0.03590587893673495 accuracy:
0.9880308508872986
Epoch 34/60: val loss: 0.0290019762718917 accuracy: 0.9893973214285714

New weights tensor([1.0000, 1.0000, 0.8819, 0.9685, 0.8895, 0.9372,
0.9030, 0.9496, 0.9173])
Learning rate 1.0000000000000004e-08
Epoch 35
avg train loss 30 iterations: 0.035746994637321734 accuracy:
0.9880906343460083
avg train loss 30 iterations: 0.03557387690086311 accuracy:
0.9881594181060791
avg train loss 30 iterations: 0.03542415089358061 accuracy:
0.988215446472168
avg train loss 30 iterations: 0.03525082468796843 accuracy:
0.9882827401161194
avg train loss 30 iterations: 0.03508916131535172 accuracy:
0.988343358039856
Epoch 35/60: val loss: 0.027981948993068988 accuracy:
0.9888392857142857
New weights tensor([1.0000, 1.0000, 0.8900, 1.0000, 0.8842, 0.9400,
0.9705, 0.8550, 0.8842])
Learning rate 1.0000000000000004e-08
Epoch 36
avg train loss 30 iterations: 0.03493656168536521 accuracy:
0.9883995652198792
```

```
avg train loss 30 iterations: 0.03479152831771229 accuracy:
0.9884588122367859
avg train loss 30 iterations: 0.03467790795899495 accuracy:
0.9884999990463257
avg train loss 30 iterations: 0.03453786398898414 accuracy:
0.9885464906692505
avg train loss 30 iterations: 0.03441793122540336 accuracy:
0.9885924458503723
Epoch 36/60: val loss: 0.055520282812722144 accuracy:
0.9827008928571429
New weights tensor([1.0000, 0.9702, 0.8948, 1.0000, 0.9447, 0.8389,
0.8536, 0.7714, 0.8445])
Learning rate 1.0000000000000004e-08
Epoch 37
avg train loss 30 iterations: 0.03431798121380229 accuracy:
0.9886171221733093
avg train loss 30 iterations: 0.034186363228745825 accuracy:
0.988667905330658
avg train loss 30 iterations: 0.03405257313952731 accuracy:
0.9887124300003052
avg train loss 30 iterations: 0.03390298091032548 accuracy:
0.9887678027153015
avg train loss 30 iterations: 0.033752133194549304 accuracy:
0.9888225197792053
Epoch 37/60: val loss: 0.03184667113912708 accuracy:
0.9916294642857143
New weights tensor([1.0000, 1.0000, 0.8571, 1.0000, 0.9426, 0.9382,
0.9420, 0.9400, 0.9400])
Learning rate 1.0000000000000004e-08
Epoch 38
avg train loss 30 iterations: 0.03364952294275044 accuracy:
0.9888563752174377
avg train loss 30 iterations: 0.03352618168886975 accuracy:
0.9889045357704163
avg train loss 30 iterations: 0.03340437406014457 accuracy:
0.9889411330223083
avg train loss 30 iterations: 0.03326782411288015 accuracy:
0.9889992475509644
avg train loss 30 iterations: 0.033127166156526194 accuracy:
0.9890567660331726
Epoch 38/60: val loss: 0.030063057323007212 accuracy:
0.9877232142857143
New weights tensor([1.0000, 1.0000, 0.9107, 1.0000, 0.9322, 0.9478,
0.9409, 0.7767, 0.8635])
Learning rate 1.0000000000000004e-08
Epoch 39
avg train loss 30 iterations: 0.0329913229605825 accuracy:
0.9891101717948914
avg train loss 30 iterations: 0.03289390472794942 accuracy:
```

```
0.9891557693481445
avg train loss 30 iterations: 0.032771506282492056 accuracy:
0.9891954660415649
avg train loss 30 iterations: 0.03264727104437268 accuracy:
0.9892454743385315
avg train loss 30 iterations: 0.03256397328871809 accuracy:
0.989284336566925
Epoch 39/60: val loss: 0.044844869967294344 accuracy:
0.9849330357142857
New weights tensor([0.9406, 1.0000, 0.9051, 1.0000, 0.9348, 0.8925,
0.9699, 0.8382, 0.7699])
Learning rate 1.0000000000000004e-08
Epoch 40
avg train loss 30 iterations: 0.03245363587573829 accuracy:
0.9893193244934082
avg train loss 30 iterations: 0.032327717624254494 accuracy:
0.9893732070922852
avg train loss 30 iterations: 0.032213970151858236 accuracy:
0.9894160628318787
avg train loss 30 iterations: 0.03208681634038096 accuracy:
0.989463746547699
avg train loss 30 iterations: 0.03197594440052635 accuracy:
0.9895005226135254
Epoch 40/60: val loss: 0.03991971376568212 accuracy: 0.98828125
New weights tensor([1.0000, 0.9688, 0.9129, 1.0000, 0.9704, 0.9409,
0.9445, 0.7138, 0.9447])
Learning rate 1.0000000000000004e-08
Epoch 41
avg train loss 30 iterations: 0.03187818964406283 accuracy:
0.9895387291908264
avg train loss 30 iterations: 0.031826782363264625 accuracy:
0.9895440340042114
avg train loss 30 iterations: 0.031746190018229474 accuracy:
0.9895697236061096
avg train loss 30 iterations: 0.0316723014468247 accuracy:
0.9895901083946228
avg train loss 30 iterations: 0.031630550543510584 accuracy:
0.9895951151847839
Epoch 41/60: val loss: 0.002147171240981801 accuracy:
0.9994419642857143
New weights tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.9696,
1.0000, 1.0000, 1.0000])
Learning rate 1.0000000000000004e-08
Epoch 42
avg train loss 30 iterations: 0.031599498708070765 accuracy:
0.989601731300354
avg train loss 30 iterations: 0.03157156905461297 accuracy:
0.9896166324615479
avg train loss 30 iterations: 0.031541287103483286 accuracy:
```

```
0.9896264672279358
avg train loss 30 iterations: 0.03152457952527963 accuracy:
0.9896262288093567
avg train loss 30 iterations: 0.03145751113922268 accuracy:
0.9896506667137146
Epoch 42/60: val loss: 0.002485955114544985 accuracy: 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.])
Learning rate 1.0000000000000004e-08
Epoch 43
avg train loss 30 iterations: 0.03140370193726462 accuracy:
0.9896765351295471
avg train loss 30 iterations: 0.03134541757462648 accuracy:
0.9896956086158752
avg train loss 30 iterations: 0.03127902760805418 accuracy:
0.9897193908691406
avg train loss 30 iterations: 0.031211660072059978 accuracy:
0.9897477626800537
avg train loss 30 iterations: 0.031168571463899935 accuracy:
0.9897662401199341
Epoch 43/60: val loss: 0.007438339066928685 accuracy:
0.9983258928571429
New weights tensor([1.0000, 1.0000, 0.9695, 1.0000, 1.0000, 1.0000,
1.0000, 0.9720, 0.9707])
Learning rate 1.0000000000000004e-08
Epoch 44
avg train loss 30 iterations: 0.03106963942428946 accuracy:
0.9898005127906799
avg train loss 30 iterations: 0.030965300647994857 accuracy:
0.989842414855957
avg train loss 30 iterations: 0.030864576752195966 accuracy:
0.9898744821548462
avg train loss 30 iterations: 0.030793520005537763 accuracy:
0.9898967742919922
avg train loss 30 iterations: 0.030686799428741977 accuracy:
0.9899377226829529
Epoch 44/60: val loss: 0.026978685261773144 accuracy:
0.9905133928571429
New weights tensor([1.0000, 1.0000, 0.9112, 0.9687, 0.8885, 0.9378,
0.9344, 0.9013, 0.9716])
Learning rate 1.0000000000000004e-08
Epoch 45
avg train loss 30 iterations: 0.03057469454793802 accuracy:
0.989975094795227
avg train loss 30 iterations: 0.030493005413266612 accuracy:
0.9900013208389282
avg train loss 30 iterations: 0.030409555489486664 accuracy:
0.990031898021698
avg train loss 30 iterations: 0.03032020457999634 accuracy:
0.9900622963905334
```

```
avg train loss 30 iterations: 0.030253412350179983 accuracy:
0.9900877475738525
Epoch 45/60: val loss: 0.029337969333315794 accuracy:
0.9916294642857143
New weights tensor([1.0000, 1.0000, 0.9161, 1.0000, 0.9120, 0.9397,
0.9707, 0.8825, 0.9403])
Learning rate 1.0000000000000004e-08
Epoch 46
avg train loss 30 iterations: 0.030177916946489153 accuracy:
0.9901190400123596
avg train loss 30 iterations: 0.03009243540268919 accuracy:
0.9901531934738159
avg train loss 30 iterations: 0.029994525467822553 accuracy:
0.9901869893074036
avg train loss 30 iterations: 0.029928964631363787 accuracy:
0.9902114868164062
avg train loss 30 iterations: 0.029845064575025407 accuracy:
0.9902492761611938
Epoch 46/60: val loss: 0.05261567253806528 accuracy:
0.9827008928571429
New weights tensor([1.0000, 0.9702, 0.8683, 1.0000, 0.9184, 0.8915,
0.8825, 0.7681, 0.8169])
Learning rate 1.0000000000000004e-08
Epoch 47
avg train loss 30 iterations: 0.029745146681261343 accuracy:
0.9902881383895874
avg train loss 30 iterations: 0.029650495240096483 accuracy:
0.9903163313865662
avg train loss 30 iterations: 0.02955849935463408 accuracy:
0.990348756313324
avg train loss 30 iterations: 0.02946986742759465 accuracy:
0.9903852939605713
avg train loss 30 iterations: 0.0293852633541498 accuracy:
0.990412712097168
Epoch 47/60: val loss: 0.027924146283891917 accuracy: 0.98828125
New weights tensor([1.0000, 1.0000, 0.8831, 1.0000, 0.9423, 0.9080,
0.9153, 0.8298, 0.9107])
Learning rate 1.0000000000000004e-08
Epoch 48
avg train loss 30 iterations: 0.029283342798717544 accuracy:
0.9904544353485107
avg train loss 30 iterations: 0.029191204242927517 accuracy:
0.9904900789260864
avg train loss 30 iterations: 0.029091226727550205 accuracy:
0.9905297756195068
avg train loss 30 iterations: 0.029007026339360004 accuracy:
0.9905648231506348
avg train loss 30 iterations: 0.02894535483689856 accuracy:
0.9905866384506226
```

```
Epoch 48/60: val loss: 0.03048274679739344 accuracy:
0.9888392857142857
New weights tensor([1.0000, 1.0000, 0.8831, 1.0000, 0.8676, 0.8984,
0.9699, 0.9116, 0.8890])
Learning rate 1.0000000000000004e-08
Epoch 49
avg train loss 30 iterations: 0.028857277087651095 accuracy:
0.9906138181686401
avg train loss 30 iterations: 0.0287835180914738 accuracy:
0.9906395673751831
avg train loss 30 iterations: 0.028709717082123158 accuracy:
0.990669310092926
avg train loss 30 iterations: 0.028629368748965673 accuracy:
0.9906945824623108
avg train loss 30 iterations: 0.0285370683942937 accuracy:
0.9907323122024536
Epoch 49/60: val loss: 0.03738170445181562 accuracy:
0.9871651785714286
New weights tensor([0.9125, 1.0000, 0.9359, 1.0000, 0.9355, 0.8425,
0.9698, 0.8382, 0.9169])
Learning rate 1.0000000000000004e-08
Epoch 50
avg train loss 30 iterations: 0.028463668117165632 accuracy:
0.9907583594322205
avg train loss 30 iterations: 0.028366344734587916 accuracy:
0.9907954931259155
avg train loss 30 iterations: 0.02830009600216935 accuracy:
0.9908198714256287
avg train loss 30 iterations: 0.028211148507738072 accuracy:
0.9908564686775208
avg train loss 30 iterations: 0.028125091093900587 accuracy:
0.99088454246521
Epoch 50/60: val loss: 0.03655927968481722 accuracy:
0.9905133928571429
New weights tensor([1.0000, 0.9690, 0.9418, 1.0000, 0.9415, 0.9120,
0.9442, 0.7903, 1.0000])
Learning rate 1.0000000000000004e-08
Epoch 51
avg train loss 30 iterations: 0.028065511790780893 accuracy:
0.9909053444862366
avg train loss 30 iterations: 0.028027483099293204 accuracy:
0.9909165501594543
avg train loss 30 iterations: 0.028032122145238373 accuracy:
0.9909195303916931
avg train loss 30 iterations: 0.02797227140137048 accuracy:
0.9909387230873108
avg train loss 30 iterations: 0.02793474441325419 accuracy:
0.9909496903419495
Epoch 51/60: val loss: 0.003648816655446093 accuracy:
```

```
0.9988839285714286
New weights tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.9696,
1.0000, 1.0000, 0.9695])
Learning rate 1.0000000000000004e-08
Epoch 52
avg train loss 30 iterations: 0.027920609424653863 accuracy:
0.9909495711326599
avg train loss 30 iterations: 0.02791510812160371 accuracy:
0.9909523725509644
avg train loss 30 iterations: 0.027876896350663507 accuracy:
0.9909631013870239
avg train loss 30 iterations: 0.027825494231730766 accuracy:
0.9909818172454834
avg train loss 30 iterations: 0.02776805351360489 accuracy:
0.9910082817077637
Epoch 52/60: val loss: 0.001862324222331933 accuracy: 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.])
Learning rate 1.0000000000000004e-08
Epoch 53
avg train loss 30 iterations: 0.02771698133066804 accuracy:
0.99102783203125
avg train loss 30 iterations: 0.0277191393834772 accuracy:
0.991034209728241
avg train loss 30 iterations: 0.02765211564227808 accuracy:
0.991060197353363
avg train loss 30 iterations: 0.02759999753985887 accuracy:
0.9910860061645508
avg train loss 30 iterations: 0.027547246264123575 accuracy:
0.9911037683486938
Epoch 53/60: val loss: 0.006177911033124214 accuracy:
0.9983258928571429
New weights tensor([1.0000, 1.0000, 0.9409, 1.0000, 1.0000, 1.0000,
0.9711, 1.0000, 1.0000])
Learning rate 1.0000000000000004e-08
Epoch 54
avg train loss 30 iterations: 0.027461018669410295 accuracy:
0.9911303520202637
avg train loss 30 iterations: 0.02739542117487457 accuracy:
0.9911594390869141
avg train loss 30 iterations: 0.027321888873220745 accuracy:
0.9911883473396301
avg train loss 30 iterations: 0.02727063201532754 accuracy:
0.9912093877792358
avg train loss 30 iterations: 0.02720270471999755 accuracy:
0.9912416934967041
Epoch 54/60: val loss: 0.02746791510435287 accuracy:
0.9910714285714286
New weights tensor([1.0000, 1.0000, 0.8831, 0.9687, 0.9161, 0.9375,
0.9348, 0.9005, 1.0000])
```

```
Learning rate 1.0000000000000004e-08
Epoch 55
avg train loss 30 iterations: 0.027123135262530424 accuracy:
0.991274893283844
avg train loss 30 iterations: 0.027048790369009975 accuracy:
0.9913029670715332
avg train loss 30 iterations: 0.026984273306253283 accuracy:
0.9913270473480225
avg train loss 30 iterations: 0.026928055207708235 accuracy:
0.9913547039031982
avg train loss 30 iterations: 0.02686822051164585 accuracy:
0.9913784265518188
Epoch 55/60: val loss: 0.027179117395072745 accuracy:
0.9916294642857143
New weights tensor([1.0000, 1.0000, 0.8900, 1.0000, 0.9406, 0.9116,
0.9415, 0.9696, 0.9116])
Learning rate 1.0000000000000004e-08
Epoch 56
avg train loss 30 iterations: 0.026795988900654842 accuracy:
0.991410493850708
avg train loss 30 iterations: 0.026762679437547253 accuracy:
0.9914188385009766
avg train loss 30 iterations: 0.02669849817140057 accuracy:
0.991445779800415
avg train loss 30 iterations: 0.02662813743208162 accuracy:
0.9914762377738953
avg train loss 30 iterations: 0.026552285392612267 accuracy:
0.9915065169334412
Epoch 56/60: val loss: 0.05624570334371778 accuracy: 0.984375
New weights tensor([1.0000, 0.9704, 0.8934, 1.0000, 0.9720, 0.8654,
0.8543, 0.7703, 0.8745])
Learning rate 1.0000000000000004e-08
Epoch 57
avg train loss 30 iterations: 0.026478028287249294 accuracy:
0.9915338754653931
avg train loss 30 iterations: 0.02641963661037713 accuracy:
0.9915563464164734
avg train loss 30 iterations: 0.026362151941264376 accuracy:
0.9915750026702881
avg train loss 30 iterations: 0.026308552829172483 accuracy:
0.9915899038314819
avg train loss 30 iterations: 0.026254148749053387 accuracy:
0.9916119575500488
Epoch 57/60: val loss: 0.03208563581003026 accuracy:
0.9893973214285714
New weights tensor([1.0000, 1.0000, 0.8825, 1.0000, 0.9149, 0.9378,
0.9157, 0.8831, 0.9107])
Learning rate 1.0000000000000004e-08
Epoch 58
```

```
avg train loss 30 iterations: 0.02619151140085263 accuracy:
0.9916348457336426
avg train loss 30 iterations: 0.02614494540490452 accuracy:
0.9916529655456543
avg train loss 30 iterations: 0.026089961810427928 accuracy:
0.991674542427063
avg train loss 30 iterations: 0.026046537477795752 accuracy:
0.9916888475418091
avg train loss 30 iterations: 0.02601756840311949 accuracy:
0.9916995167732239
Epoch 58/60: val loss: 0.029370290726676882 accuracy:
0.9905133928571429
New weights tensor([1.0000, 1.0000, 0.9107, 1.0000, 0.9658, 0.9226,
0.9409, 0.8836, 0.8885])
Learning rate 1.0000000000000004e-08
Epoch 59
avg train loss 30 iterations: 0.02595576216797054 accuracy:
0.9917216897010803
avg train loss 30 iterations: 0.025888939177616692 accuracy:
0.9917463064193726
avg train loss 30 iterations: 0.025824619542147795 accuracy:
0.9917672276496887
avg train loss 30 iterations: 0.025775282712656215 accuracy:
0.9917809367179871
avg train loss 30 iterations: 0.025723434388885323 accuracy:
0.9917945861816406
Epoch 59/60: val loss: 0.03794878405278723 accuracy: 0.98828125
New weights tensor([0.9701, 1.0000, 0.9359, 1.0000, 0.9046, 0.8910,
0.9699, 0.8135, 0.9169])
Learning rate 1.0000000000000004e-08
Epoch 60
avg train loss 30 iterations: 0.025666430421749306 accuracy:
0.9918195605278015
avg train loss 30 iterations: 0.025623581068319683 accuracy:
0.991832971572876
avg train loss 30 iterations: 0.02557097517523357 accuracy:
0.9918567538261414
avg train loss 30 iterations: 0.0255109565156149 accuracy:
0.9918803572654724
avg train loss 30 iterations: 0.02547230269611184 accuracy:
0.9918864965438843
Epoch 60/60: val loss: 0.03816026020649588 accuracy:
0.9877232142857143
New weights tensor([1.0000, 0.9388, 0.9137, 1.0000, 0.9701, 0.9107,
0.8660, 0.7914, 0.9720])
Learning rate 1.0000000000000004e-08

train done
```

```
model.save("best")
model.load("best")

Downloading...
From (original): https://drive.google.com/uc?id=1-
KG_C4OaeYnh9DpB4GH0_fdUoLK706Fg
From (redirected): https://drive.google.com/uc?id=1-
KG_C4OaeYnh9DpB4GH0_fdUoLK706Fg&confirm=t&uuid=a1315c98-8002-48c5-
b778-14fbd56f8e59
To: /content/best.pth
100%|██████████| 44.8M/44.8M [00:00<00:00, 80.8MB/s]
<ipython-input-65-62702a8d8649>:18: FutureWarning: You are using
`torch.load` with `weights_only=False` (the current default value),
which uses the default pickle module implicitly. It is possible to
construct malicious pickle data which will execute arbitrary code
during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-
models for more details). In a future release, the default value for
`weights_only` will be flipped to `True`. This limits the functions
that could be executed during unpickling. Arbitrary objects will no
longer be allowed to be loaded via this mode unless they are
explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control
of the loaded file. Please open an issue on GitHub for any issues
related to this experimental feature.
  self.model.load_state_dict(torch.load(output))
```

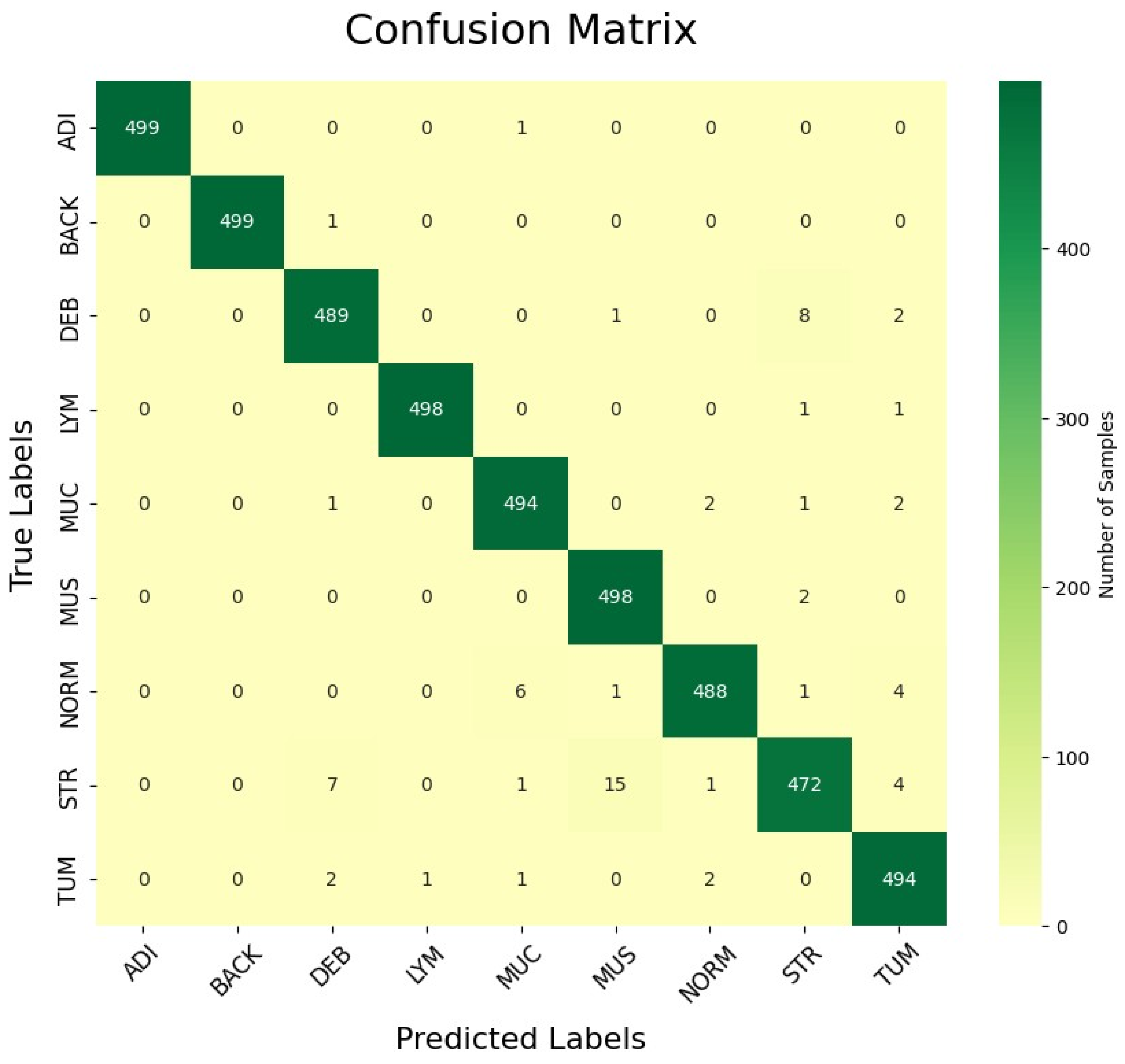
Пример тестирования модели на части набора данных:

```
# evaluating model on 10% of test dataset
 #(чет вообще все плохо с конф. матрицей, даже не знаю почему... но на
другом тесте работает нормальноЫ)
pred_1 = model.test_on_dataset(d_test, limit=0.1)
#confi_matrix(d_test.labels[:len(pred_1)], pred_1, TISSUE_CLASSES)
Metrics.print_all(d_test.labels[:len(pred_1)], pred_1, '10% of test')

{"model_id":"41564e70d36d4237bb25dea0959e5e82","version_major":2,"vers
ion_minor":0}

metrics for 10% of test:
	accuracy 0.9978:
	balanced accuracy 0.9978:

/usr/local/lib/python3.10/dist-packages/sklearn/metrics/
_classification.py:2480: UserWarning: y_pred contains classes not in
y_true
  warnings.warn("y_pred contains classes not in y_true")
```

Пример тестирования модели на полном наборе данных:

```
# evaluating model on full test dataset (may take time)
if TEST_ON_LARGE_DATASET:
    pred_2 = model.test_on_dataset(d_test)
    confi_matrix(d_test.labels, pred_2,  TISSUE_CLASSES)
    Metrics.print_all(d_test.labels, pred_2, 'test')
```

{"model_id":"f0aa6369431044f393d732340019e716","version_major":2,"version_minor":0}



	ADI	BACK	DEB	LYM	MUC	MUS	NORM	STR	TUM
ADI	499	0	0	0	1	0	0	0	0
BACK	0	499	1	0	0	0	0	0	0
DEB	0	0	489	0	0	1	0	8	2
LYM	0	0	0	498	0	0	0	1	1
MUC	0	0	1	0	494	0	2	1	2
MUS	0	0	0	0	0	498	0	2	0
NORM	0	0	0	0	6	1	488	1	4
STR	0	0	7	0	1	15	1	472	4
TUM	0	0	2	1	1	0	2	0	494

```
Class          True Positive (TP)  False Positive (FP)
-------------------------------------------------------
ADI            499                 0
BACK           499                 0
DEB            489                 11
```

```
LYM            498               1
MUC            494               9
MUS            498               17
NORM           488               5
STR            472               13
TUM            494               13
metrics for test:
	 accuracy 0.9847:
	 balanced accuracy 0.9847:
```

Результат работы пайплайна обучения и тестирования выше тоже будет оцениваться. Поэтому не забудьте присылать на проверку ноутбук с выполнеными 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 = 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 'pretrained' is deprecated
since 0.13 and may be removed in the future, please use 'weights'
instead.
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:2
23: UserWarning: Arguments other than a weight enum or `None` for
'weights' are deprecated since 0.13 and may be removed in the future.
The current behavior is equivalent to passing
`weights=ResNet18_Weights.IMAGENET1K_V1`. You can also use
`weights=ResNet18_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
Downloading...
```

```
From (original): https://drive.google.com/uc?id=1-
KG_C4OaeYnh9DpB4GH0_fdUoLK706Fg
From (redirected): https://drive.google.com/uc?id=1-
KG_C4OaeYnh9DpB4GH0_fdUoLK706Fg&confirm=t&uuid=33e5016e-feda-4ce4-
95db-72b9eb7f4591
To: /content/best.pth
100%|██████████| 44.8M/44.8M [00:00<00:00, 75.3MB/s]
<ipython-input-65-62702a8d8649>:18: FutureWarning: You are using
`torch.load` with `weights_only=False` (the current default value),
which uses the default pickle module implicitly. It is possible to
construct malicious pickle data which will execute arbitrary code
during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-
models for more details). In a future release, the default value for
`weights_only` will be flipped to `True`. This limits the functions
that could be executed during unpickling. Arbitrary objects will no
longer be allowed to be loaded via this mode unless they are
explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control
of the loaded file. Please open an issue on GitHub for any issues
related to this experimental feature.
  self.model.load_state_dict(torch.load(output))
Downloading...
From: https://drive.google.com/uc?
export=download&confirm=pbef&id=1F0ve7Cl7c7Ln-7hoQLbFauaAjsVAh1g7
To: /content/test_tiny.npz
100%|██████████| 10.6M/10.6M [00:00<00:00, 54.7MB/s]

Loading dataset test_tiny from npz.
Done. Dataset test_tiny consists of 90 images.

{"model_id":"936545840aff43a5bb13cfba2d8078b3","version_major":2,"vers
ion_minor":0}

metrics for test-tiny:
	accuracy 0.9667:
	balanced accuracy 0.9667:
```

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.

```python
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 нейроной сети, использующейся для классификации изображений из набора данных MNIST.

```python
# 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 туториалов 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 удлось сразу разобраться или хочется вникнуть в тему более глубоко), можете посмотреть следующий отличный туториал: 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/cbernet/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 осущетвляться 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"
```

```
----------------------------------------------------------------------
-----
FileNotFoundError                         Traceback (most recent call
last)
<ipython-input-16-e0c49c38d470> in <cell line: 10>()
      9
     10 if not (Path(p) / 'tmp').exists():
---> 11     (Path(p) / 'tmp').mkdir()
     12
     13 img1.save(str(Path(p) / 'tmp' / 'img1.png'))

/usr/lib/python3.10/pathlib.py in mkdir(self, mode, parents, exist_ok)
   1173         """
   1174         try:
-> 1175             self._accessor.mkdir(self, mode)
   1176         except FileNotFoundError:
   1177             if not parents or self.parent == self:

FileNotFoundError: [Errno 2] No such file or directory:
'/content/drive/MyDrive/dev/prak_nn_1/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"

[Errno 2] No such file or directory:
'/content/drive/MyDrive//dev/prak_nn_1/'
/content
unzip:  cannot find or open tmp.zip, tmp.zip.zip or tmp.zip.ZIP.

from google.colab import drive
drive.mount('/content/drive')
```