#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
from IPython.display import clear_output
```

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
```

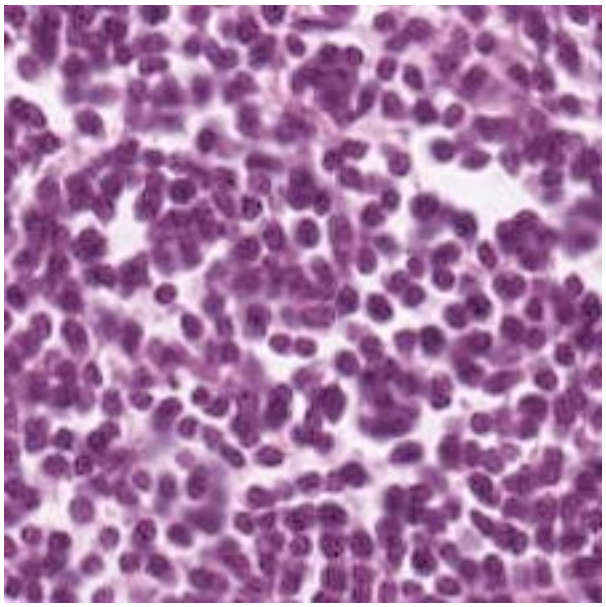
```
        self.is_loaded = False
        url = f"https://drive.google.com/uc?
export=download&confirm=pbef&id={DATASETS_LINKS[name]}"
        output = f'{name}.npz'
        gdown.download(url, output, quiet=False)
        print(f'Loading dataset {self.name} from npz.')
        np_obj = np.load(f'{name}.npz')
        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.

```
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, 127MB/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 3.
Label code corresponds to LYM 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

```
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=''):
    clear_output(wait=True)
    fig, axes = plt.subplots(1, 2, figsize=(15, 5))
```

```
    axes[0].set_title("Loss")
    axes[0].plot(train_loss, label="train")
    axes[0].plot(val_loss, label="validation")
    axes[0].legend()

    axes[1].set_title("Validation accuracy")
    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',
            "best_model": '1L9PiS1JbKKOnxaGleh85wfz_yP5lf0H8',
            "best_1": '1-3gFYm4uEM4eOg8niDPqdNYICLobtMID'
        }
        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):
        '''
        param_grid =
        {
            "learning_rate": [1e-3, 1e-4],
            "weight_decay": [1e-4, 1e-5],
            "batch_size": [32, 64],
        }

        best_val_accuracy = 0.0
        best_params = None

        for params in ParameterGrid(param_grid):
            print(f"Training with params: {params}")
            learning_rate = params["learning_rate"]
            weight_decay = params["weight_decay"]
            batch_size = params["batch_size"]
        '''
        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")

        #LBL11
        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)

        best_val_accuracy = 0.0
        best_model_path = '/content/drive/MyDrive/DP/best_model.pth'

        #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)

            if val_accuracy > best_val_accuracy:
                best_val_accuracy = val_accuracy
                torch.save(self.model.state_dict(), best_model_path)
                print(f"New best model saved with accuracy:
{best_val_accuracy}")


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'.

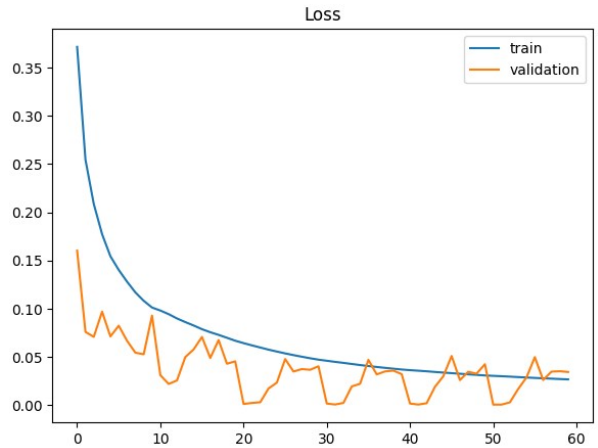
```
d_train = Dataset("train")
d_test = Dataset("test")
```

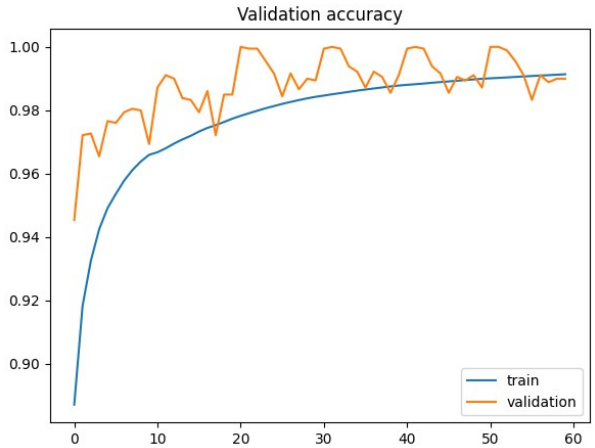
```
Downloading...
From: https://drive.google.com/uc?
export=download&confirm=pbef&id=1T5z6OPLoYUANn-_B14PSngZFm0mIOgsU
To: /content/train.npz
100%|██████████| 2.10G/2.10G [00:26<00:00, 80.8MB/s]

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, 108MB/s]

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

```
model = Model()
```

```
/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: "https://download.pytorch.org/models/resnet18-
f37072fd.pth" to /root/.cache/torch/hub/checkpoints/resnet18-
f37072fd.pth
100%|██████████| 44.7M/44.7M [00:00<00:00, 157MB/s]
```

```python
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_1")
```






```
Epoch 1
avg train loss 30 iterations: 0.8339256688952446 accuracy:
0.7583333253860474
avg train loss 30 iterations: 0.5934630331893762 accuracy:
0.824999988079071
avg train loss 30 iterations: 0.4790039273599784 accuracy:
0.8541666865348816
```

```
avg train loss 30 iterations: 0.4253099443390965 accuracy:
0.8695312738418579
avg train loss 30 iterations: 0.37279683654507 accuracy:
0.8866666555404663
Epoch 1/60: val loss: 0.16028010577429086 accuracy: 0.9453125
New weights tensor([0.9004, 0.9709, 0.5025, 0.8683, 0.8167, 0.7531,
0.8306, 0.5158, 0.4508])
Learning rate 0.0001
Epoch 2
avg train loss 30 iterations: 0.33676408119088047 accuracy:
0.8957182168960571
avg train loss 30 iterations: 0.3123400632062512 accuracy:
0.902103066444397
avg train loss 30 iterations: 0.2879948101597703 accuracy:
0.908713698387146
avg train loss 30 iterations: 0.2677210097328323 accuracy:
0.91501384973526
avg train loss 30 iterations: 0.2550497363473094 accuracy:
0.917981743812561
Epoch 2/60: val loss: 0.07606179182351168 accuracy: 0.9720982142857143

New weights tensor([1.0000, 0.9716, 0.8040, 0.9418, 1.0000, 0.9412,
0.8617, 0.5627, 0.5899])
Learning rate 0.0001
Epoch 3
avg train loss 30 iterations: 0.24185414662786636 accuracy:
0.9222515225410461
avg train loss 30 iterations: 0.23053346253231743 accuracy:
0.9258459806442261
avg train loss 30 iterations: 0.22320626633792964 accuracy:
0.9284119606018066
avg train loss 30 iterations: 0.21644962626240125 accuracy:
0.930168867111206
avg train loss 30 iterations: 0.2088888952533294 accuracy:
0.9323838353157043
Epoch 3/60: val loss: 0.0708982434090493 accuracy: 0.97265625
New weights tensor([0.9129, 1.0000, 0.9120, 0.9702, 0.9199, 0.7659,
0.5579, 0.7483, 0.9412])
Learning rate 0.0001
Epoch 4
avg train loss 30 iterations: 0.20049938341066642 accuracy:
0.9350414276123047
avg train loss 30 iterations: 0.19329998159116646 accuracy:
0.9373781681060791
avg train loss 30 iterations: 0.1875912813558955 accuracy:
0.9393416047096252
avg train loss 30 iterations: 0.18239688281340435 accuracy:
0.9409903883934021
avg train loss 30 iterations: 0.17712968790048045 accuracy:
```

```
0.9425269365310669
Epoch 4/60: val loss: 0.09701563816218238 accuracy: 0.9654017857142857

New weights tensor([0.9409, 1.0000, 0.6186, 0.9688, 0.9157, 0.8232,
0.5933, 0.6133, 0.9439])
Learning rate 0.0001
Epoch 5
avg train loss 30 iterations: 0.17163844897238426 accuracy:
0.9440063238143921
avg train loss 30 iterations: 0.16658206863177336 accuracy:
0.9454066157341003
avg train loss 30 iterations: 0.16229779284544052 accuracy:
0.9466408491134644
avg train loss 30 iterations: 0.1585179678587634 accuracy:
0.9476001262664795
avg train loss 30 iterations: 0.15462696746816804 accuracy:
0.948980450630188
Epoch 5/60: val loss: 0.07132868733606301 accuracy: 0.9765625
New weights tensor([1.0000, 1.0000, 0.9165, 0.9692, 0.9701, 0.8543,
0.9141, 0.7045, 0.5628])
Learning rate 0.0001
Epoch 6
avg train loss 30 iterations: 0.1509624616438701 accuracy:
0.9501592516899109
avg train loss 30 iterations: 0.14703067059262992 accuracy:
0.9514570832252502
avg train loss 30 iterations: 0.14494416427512008 accuracy:
0.9519230723381042
avg train loss 30 iterations: 0.14231007862277328 accuracy:
0.9528214335441589
avg train loss 30 iterations: 0.1403649075962706 accuracy:
0.9535220861434937
Epoch 6/60: val loss: 0.08243654452131263 accuracy: 0.9760044642857143

New weights tensor([1.0000, 0.9702, 0.7535, 1.0000, 0.8672, 0.9445,
0.7725, 0.6951, 0.8169])
Learning rate 0.0001
Epoch 7
avg train loss 30 iterations: 0.13744897221072783 accuracy:
0.9546273946762085
avg train loss 30 iterations: 0.13492801109513008 accuracy:
0.955518901348114
avg train loss 30 iterations: 0.13243322672456107 accuracy:
0.9563880562782288
avg train loss 30 iterations: 0.13030598224249632 accuracy:
0.9571759104728699
avg train loss 30 iterations: 0.12814567439305724 accuracy:
0.9577414989471436
Epoch 7/60: val loss: 0.06677004168575097 accuracy: 0.9793526785714286
```

```
New weights tensor([1.0000, 1.0000, 0.8571, 1.0000, 0.9426, 0.8783,
0.9153, 0.7788, 0.6170])
Learning rate 0.0001
Epoch 8
avg train loss 30 iterations: 0.12542156479171485 accuracy:
0.958630383014679
avg train loss 30 iterations: 0.12335632681002842 accuracy:
0.9591540098190308
avg train loss 30 iterations: 0.12106156306066847 accuracy:
0.9598681330680847
avg train loss 30 iterations: 0.1192114245258873 accuracy:
0.9603865742683411
avg train loss 30 iterations: 0.11704803464477828 accuracy:
0.9610863924026489
Epoch 8/60: val loss: 0.05448739443506513 accuracy: 0.98046875
New weights tensor([1.0000, 1.0000, 0.9400, 1.0000, 0.8676, 0.8975,
0.9409, 0.5768, 0.8375])
Learning rate 0.0001
Epoch 9
avg train loss 30 iterations: 0.11523560271485557 accuracy:
0.9616316556930542
avg train loss 30 iterations: 0.11361301511346894 accuracy:
0.9620957970619202
avg train loss 30 iterations: 0.11172578955006043 accuracy:
0.9626588821411133
avg train loss 30 iterations: 0.1098631732537501 accuracy:
0.9632671475410461
avg train loss 30 iterations: 0.10818955311599697 accuracy:
0.9638024568557739
Epoch 9/60: val loss: 0.05285803640539858 accuracy: 0.9799107142857143

New weights tensor([0.9701, 1.0000, 0.9359, 0.9732, 0.9355, 0.9184,
0.8842, 0.6527, 0.7699])
Learning rate 0.0001
Epoch 10
avg train loss 30 iterations: 0.1072326577596247 accuracy:
0.9641153812408447
avg train loss 30 iterations: 0.10588148513005179 accuracy:
0.9645656943321228
avg train loss 30 iterations: 0.10403876588390619 accuracy:
0.9651914834976196
avg train loss 30 iterations: 0.10270070980318417 accuracy:
0.9654749631881714
avg train loss 30 iterations: 0.10136277485980465 accuracy:
0.9659128785133362
Epoch 10/60: val loss: 0.09295280559620421 accuracy:
0.9693080357142857
New weights tensor([0.9133, 1.0000, 0.8075, 1.0000, 0.9701, 0.6032,
0.6425, 0.7389, 0.8920])
```

```
Learning rate 0.0001
Epoch 11
avg train loss 30 iterations: 0.10067902999036558 accuracy:
0.9660916924476624
avg train loss 30 iterations: 0.09991451445114236 accuracy:
0.9663614630699158
avg train loss 30 iterations: 0.09914443929636035 accuracy:
0.9665820598602295
avg train loss 30 iterations: 0.09864519400353561 accuracy:
0.9666411280632019
avg train loss 30 iterations: 0.09821432693931555 accuracy:
0.9666980504989624
Epoch 11/60: val loss: 0.03116937797001031 accuracy:
0.9871651785714286
New weights tensor([0.9326, 1.0000, 0.9161, 1.0000, 0.9173, 0.9692,
0.9129, 0.7828, 0.9103])
Learning rate 0.0001
Epoch 12
avg train loss 30 iterations: 0.09723538735419918 accuracy:
0.967012882232666
avg train loss 30 iterations: 0.09642663011032458 accuracy:
0.967297375202179
avg train loss 30 iterations: 0.09574526449376085 accuracy:
0.9675363898277283
avg train loss 30 iterations: 0.09497474563456569 accuracy:
0.9677674174308777
avg train loss 30 iterations: 0.09446234566062865 accuracy:
0.9679734706878662
Epoch 12/60: val loss: 0.02207537479885754 accuracy:
0.9910714285714286
New weights tensor([1.0000, 1.0000, 0.8577, 1.0000, 0.9698, 1.0000,
0.9708, 0.9683, 0.7725])
Learning rate 0.0001
Epoch 13
avg train loss 30 iterations: 0.09347682445634654 accuracy:
0.9683428406715393
avg train loss 30 iterations: 0.09249816488025114 accuracy:
0.9686832427978516
avg train loss 30 iterations: 0.09169547408209579 accuracy:
0.9690128564834595
avg train loss 30 iterations: 0.09072037896719388 accuracy:
0.9692837595939636
avg train loss 30 iterations: 0.08999583555637526 accuracy:
0.969450831413269
Epoch 13/60: val loss: 0.025761462134791406 accuracy:
0.9899553571428571
New weights tensor([1.0000, 1.0000, 0.9120, 1.0000, 0.9457, 0.9061,
0.8890, 0.8418, 1.0000])
Learning rate 0.0001
```

```
Epoch 14
avg train loss 30 iterations: 0.08902884723396193 accuracy:
0.9697691798210144
avg train loss 30 iterations: 0.0884736092786249 accuracy:
0.9699857831001282
avg train loss 30 iterations: 0.0877447570315746 accuracy:
0.9702569246292114
avg train loss 30 iterations: 0.0870205461293163 accuracy:
0.9705352783203125
avg train loss 30 iterations: 0.08625226279594386 accuracy:
0.9707465767860413
Epoch 14/60: val loss: 0.049965980926312374 accuracy:
0.9838169642857143
New weights tensor([1.0000, 1.0000, 0.8825, 0.9687, 0.9431, 0.9683,
0.9041, 0.6501, 0.9442])
Learning rate 0.0001
Epoch 15
avg train loss 30 iterations: 0.08545448947720684 accuracy:
0.9710237979888916
avg train loss 30 iterations: 0.08459077107295113 accuracy:
0.9712511301040649
avg train loss 30 iterations: 0.08383552658047523 accuracy:
0.9715290665626526
avg train loss 30 iterations: 0.08324068354289718 accuracy:
0.9717435240745544
avg train loss 30 iterations: 0.08270532098313294 accuracy:
0.9719108939170837
Epoch 15/60: val loss: 0.05788414968784699 accuracy:
0.9832589285714286
New weights tensor([0.9696, 1.0000, 0.8900, 0.8819, 0.8842, 0.9397,
0.8869, 0.8012, 0.8836])
Learning rate 0.0001
Epoch 16
avg train loss 30 iterations: 0.08186771732000207 accuracy:
0.9721677303314209
avg train loss 30 iterations: 0.08109661593417367 accuracy:
0.9724193811416626
avg train loss 30 iterations: 0.08051104804319517 accuracy:
0.9726380109786987
avg train loss 30 iterations: 0.0796458494866343 accuracy:
0.9729690551757812
avg train loss 30 iterations: 0.07888675287298909 accuracy:
0.9732531309127808
Epoch 16/60: val loss: 0.07078769274864628 accuracy:
0.9793526785714286
New weights tensor([0.9727, 0.9701, 0.7756, 1.0000, 0.9180, 0.8905,
0.7767, 0.7462, 0.9352])
Learning rate 0.0001
Epoch 17
```

```
avg train loss 30 iterations: 0.07827413579193858 accuracy:
0.973451554775238
avg train loss 30 iterations: 0.07754851759386287 accuracy:
0.9736974835395813
avg train loss 30 iterations: 0.07693384010795819 accuracy:
0.9739500284194946
avg train loss 30 iterations: 0.07618146735388193 accuracy:
0.9742088913917542
avg train loss 30 iterations: 0.07573069327976192 accuracy:
0.9743886590003967
Epoch 17/60: val loss: 0.04902843932255304 accuracy:
0.9860491071428571
New weights tensor([1.0000, 1.0000, 0.9696, 0.9712, 0.9149, 0.9085,
0.9429, 0.6845, 0.9107])
Learning rate 0.0001
Epoch 18
avg train loss 30 iterations: 0.07513181300544958 accuracy:
0.9746221303939819
avg train loss 30 iterations: 0.07459004627888846 accuracy:
0.9747573137283325
avg train loss 30 iterations: 0.07398745530117307 accuracy:
0.9749482274055481
avg train loss 30 iterations: 0.07346414023677822 accuracy:
0.9751232862472534
avg train loss 30 iterations: 0.07297019308017663 accuracy:
0.975271463394165
Epoch 18/60: val loss: 0.06758711232188423 accuracy:
0.9720982142857143
New weights tensor([1.0000, 1.0000, 0.8831, 1.0000, 0.8369, 0.9478,
0.9125, 0.4567, 0.6976])
Learning rate 1e-05
Epoch 19
avg train loss 30 iterations: 0.07230958328603952 accuracy:
0.975482165813446
avg train loss 30 iterations: 0.07188800238136989 accuracy:
0.9756231904029846
avg train loss 30 iterations: 0.0712572752557977 accuracy:
0.9758391380310059
avg train loss 30 iterations: 0.07059794392629251 accuracy:
0.9760724902153015
avg train loss 30 iterations: 0.06998760736441427 accuracy:
0.9762682914733887
Epoch 19/60: val loss: 0.043046195518919764 accuracy:
0.9849330357142857
New weights tensor([1.0000, 1.0000, 0.9051, 1.0000, 0.9036, 0.8930,
0.9409, 0.7885, 0.8159])
Learning rate 1e-05
Epoch 20
avg train loss 30 iterations: 0.06932026140472583 accuracy:
```

```
0.9764897227287292
avg train loss 30 iterations: 0.06866220622180964 accuracy:
0.976719856262207
avg train loss 30 iterations: 0.06808130466032025 accuracy:
0.9769136309623718
avg train loss 30 iterations: 0.06750595925855502 accuracy:
0.9771140217781067
avg train loss 30 iterations: 0.06691766809414242 accuracy:
0.9773207306861877
Epoch 20/60: val loss: 0.045515655343738866 accuracy:
0.9849330357142857
New weights tensor([1.0000, 1.0000, 0.9129, 1.0000, 0.8853, 0.9116,
0.8654, 0.7389, 0.9180])
Learning rate 1e-05
Epoch 21
avg train loss 30 iterations: 0.06644735984226544 accuracy:
0.9775102734565735
avg train loss 30 iterations: 0.06591208580141099 accuracy:
0.9776988625526428
avg train loss 30 iterations: 0.06538994311732843 accuracy:
0.9778838157653809
avg train loss 30 iterations: 0.06490218981387813 accuracy:
0.9780553579330444
avg train loss 30 iterations: 0.06440598898593793 accuracy:
0.9782235622406006
Epoch 21/60: val loss: 0.0013435833321377036 accuracy: 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.])
Learning rate 1e-05
Epoch 22
avg train loss 30 iterations: 0.0639213972739076 accuracy:
0.978395402431488
avg train loss 30 iterations: 0.06348992354721139 accuracy:
0.9785573482513428
avg train loss 30 iterations: 0.0629942946469063 accuracy:
0.9787354469299316
avg train loss 30 iterations: 0.06253656958478358 accuracy:
0.9788817763328552
avg train loss 30 iterations: 0.062100932037792274 accuracy:
0.9790349006652832
Epoch 22/60: val loss: 0.0023860221552760258 accuracy:
0.9994419642857143
New weights tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
1.0000, 1.0000, 0.9687])
Learning rate 1e-05
Epoch 23
avg train loss 30 iterations: 0.06166158143603401 accuracy:
0.9791915416717529
avg train loss 30 iterations: 0.06120372108960758 accuracy:
0.979348361492157
```

```
avg train loss 30 iterations: 0.060785954531761326 accuracy:
0.9795116782188416
avg train loss 30 iterations: 0.060311866813902736 accuracy:
0.979681134223938
avg train loss 30 iterations: 0.0598827832244655 accuracy:
0.9798387289047241
Epoch 23/60: val loss: 0.003113611552180373 accuracy:
0.9994419642857143
New weights tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
1.0000, 0.9722, 1.0000])
Learning rate 1e-05
Epoch 24
avg train loss 30 iterations: 0.05940445958245515 accuracy:
0.979999303817749
avg train loss 30 iterations: 0.058967302252514515 accuracy:
0.9801425933837891
avg train loss 30 iterations: 0.058519786434030684 accuracy:
0.980309784412384
avg train loss 30 iterations: 0.05808169810869316 accuracy:
0.9804567694664001
avg train loss 30 iterations: 0.057669087619681834 accuracy:
0.9806100130081177
Epoch 24/60: val loss: 0.0174085433037752 accuracy: 0.9955357142857143

New weights tensor([1.0000, 1.0000, 1.0000, 0.9687, 0.9712, 0.9378,
0.9668, 0.9745, 0.9439])
Learning rate 1e-05
Epoch 25
avg train loss 30 iterations: 0.05723226955651935 accuracy:
0.9807659387588501
avg train loss 30 iterations: 0.05683263809428764 accuracy:
0.9808971285820007
avg train loss 30 iterations: 0.05645977476867258 accuracy:
0.9810345768928528
avg train loss 30 iterations: 0.056083229138805 accuracy:
0.9811615347862244
avg train loss 30 iterations: 0.055688834319198216 accuracy:
0.9812946915626526
Epoch 25/60: val loss: 0.023464560717651954 accuracy:
0.9916294642857143
New weights tensor([1.0000, 1.0000, 0.9165, 1.0000, 0.9403, 0.9400,
0.9705, 0.8825, 0.9116])
Learning rate 1e-05
Epoch 26
avg train loss 30 iterations: 0.05527798190066449 accuracy:
0.9814471006393433
avg train loss 30 iterations: 0.05487033399413133 accuracy:
0.9815922379493713
avg train loss 30 iterations: 0.05450593235512881 accuracy:
```

```
0.9817270636558533
avg train loss 30 iterations: 0.05413942202574194 accuracy:
0.9818597435951233
avg train loss 30 iterations: 0.05376356483266576 accuracy:
0.9819904565811157
Epoch 26/60: val loss: 0.04788519133476906 accuracy: 0.984375
New weights tensor([1.0000, 1.0000, 0.8214, 1.0000, 0.9184, 0.9173,
0.8550, 0.8223, 0.8739])
Learning rate 1e-05
Epoch 27
avg train loss 30 iterations: 0.05338757176534449 accuracy:
0.9821157455444336
avg train loss 30 iterations: 0.05302697753333543 accuracy:
0.9822503924369812
avg train loss 30 iterations: 0.05269144263734034 accuracy:
0.9823673963546753
avg train loss 30 iterations: 0.05234002696305662 accuracy:
0.9824826717376709
avg train loss 30 iterations: 0.05198993974306843 accuracy:
0.9826039671897888
Epoch 27/60: val loss: 0.03496211967909143 accuracy:
0.9916294642857143
New weights tensor([1.0000, 1.0000, 0.9406, 1.0000, 0.9423, 0.9382,
0.9153, 0.8831, 0.9394])
Learning rate 1.0000000000000002e-06
Epoch 28
avg train loss 30 iterations: 0.05162563154730272 accuracy:
0.982735276222229
avg train loss 30 iterations: 0.05129456017465434 accuracy:
0.9828453660011292
avg train loss 30 iterations: 0.05095485148206585 accuracy:
0.9829688668251038
avg train loss 30 iterations: 0.05061351018732771 accuracy:
0.9830831289291382
avg train loss 30 iterations: 0.05031409253847665 accuracy:
0.9831958413124084
Epoch 28/60: val loss: 0.03752570974393166 accuracy:
0.9866071428571429
New weights tensor([1.0000, 1.0000, 0.9112, 1.0000, 0.8998, 0.9474,
0.8306, 0.8040, 0.9157])
Learning rate 1.0000000000000002e-06
Epoch 29
avg train loss 30 iterations: 0.0499757237662877 accuracy:
0.9833181500434875
avg train loss 30 iterations: 0.04964790957357234 accuracy:
0.9834348559379578
avg train loss 30 iterations: 0.049340235544899344 accuracy:
0.9835427403450012
avg train loss 30 iterations: 0.04903285335133642 accuracy:
```

```
0.9836490750312805
avg train loss 30 iterations: 0.04873368740829725 accuracy:
0.9837468862533569
Epoch 29/60: val loss: 0.03677860711676268 accuracy:
0.9899553571428571
New weights tensor([1.0000, 1.0000, 0.9355, 0.9732, 0.8725, 0.9184,
0.9125, 0.9434, 0.9173])
Learning rate 1.0000000000000002e-06
Epoch 30
avg train loss 30 iterations: 0.048449785303959665 accuracy:
0.9838398694992065
avg train loss 30 iterations: 0.04813683734697016 accuracy:
0.9839490652084351
avg train loss 30 iterations: 0.0478853531017264 accuracy:
0.9840148687362671
avg train loss 30 iterations: 0.04758166193528587 accuracy:
0.9841214418411255
avg train loss 30 iterations: 0.047286265656013335 accuracy:
0.984226644039154
Epoch 30/60: val loss: 0.040407853777521395 accuracy:
0.9893973214285714
New weights tensor([0.9705, 1.0000, 0.8853, 1.0000, 0.9702, 0.9400,
0.9445, 0.7389, 1.0000])
Learning rate 1.0000000000000002e-06
Epoch 31
avg train loss 30 iterations: 0.04701838227637812 accuracy:
0.9843133091926575
avg train loss 30 iterations: 0.04680874844261205 accuracy:
0.9843885898590088
avg train loss 30 iterations: 0.04661643597196515 accuracy:
0.9844561815261841
avg train loss 30 iterations: 0.046400175602741726 accuracy:
0.9845295548439026
avg train loss 30 iterations: 0.04613871930333618 accuracy:
0.984622061252594
Epoch 31/60: val loss: 0.0017615194283280289 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-06
Epoch 32
avg train loss 30 iterations: 0.04590540313379938 accuracy:
0.9846967458724976
avg train loss 30 iterations: 0.04569496542386471 accuracy:
0.9847737550735474
avg train loss 30 iterations: 0.04545380338838127 accuracy:
0.9848695397377014
avg train loss 30 iterations: 0.04522337930436446 accuracy:
0.9849445223808289
avg train loss 30 iterations: 0.04498498550758754 accuracy:
```

```
0.9850315451622009
Epoch 32/60: val loss: 0.0007922344797667133 accuracy: 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.])
Learning rate 1.0000000000000002e-06
Epoch 33
avg train loss 30 iterations: 0.04475826695459468 accuracy:
0.9851141571998596
avg train loss 30 iterations: 0.04454415219660065 accuracy:
0.9851862788200378
avg train loss 30 iterations: 0.0443791221155261 accuracy:
0.9852448105812073
avg train loss 30 iterations: 0.04416698529485001 accuracy:
0.985321581363678
avg train loss 30 iterations: 0.04393142503063339 accuracy:
0.9854037165641785
Epoch 33/60: val loss: 0.002421303220476797 accuracy:
0.9994419642857143
New weights tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
0.9714, 1.0000, 1.0000])
Learning rate 1.0000000000000002e-06
Epoch 34
avg train loss 30 iterations: 0.043709227939294076 accuracy:
0.9854752421379089
avg train loss 30 iterations: 0.043472202076705484 accuracy:
0.9855616688728333
avg train loss 30 iterations: 0.0432332885654986 accuracy:
0.9856470823287964
avg train loss 30 iterations: 0.043024221552326034 accuracy:
0.9857192039489746
avg train loss 30 iterations: 0.042809475190447886 accuracy:
0.985790491104126
Epoch 34/60: val loss: 0.0194784358344415 accuracy: 0.9938616071428571

New weights tensor([1.0000, 1.0000, 0.9394, 0.9685, 0.9434, 0.9682,
0.9668, 0.9745, 0.9173])
Learning rate 1.0000000000000002e-06
Epoch 35
avg train loss 30 iterations: 0.04257918020069963 accuracy:
0.9858636856079102
avg train loss 30 iterations: 0.04235546357359139 accuracy:
0.9859453439712524
avg train loss 30 iterations: 0.04213667199654509 accuracy:
0.9860260486602783
avg train loss 30 iterations: 0.04191352680639779 accuracy:
0.9861057996749878
avg train loss 30 iterations: 0.04169509695546134 accuracy:
0.9861788153648376
Epoch 35/60: val loss: 0.022276641994722013 accuracy: 0.9921875
New weights tensor([0.9693, 1.0000, 0.9437, 1.0000, 0.9406, 0.9400,
```

```
0.9705, 0.8825, 0.9406])
Learning rate 1.0000000000000002e-06
Epoch 36
avg train loss 30 iterations: 0.04147479255251653 accuracy:
0.9862534999847412
avg train loss 30 iterations: 0.04126063170598207 accuracy:
0.9863306879997253
avg train loss 30 iterations: 0.04107927829185149 accuracy:
0.9863837361335754
avg train loss 30 iterations: 0.040876288774269465 accuracy:
0.9864535331726074
avg train loss 30 iterations: 0.04067195099862947 accuracy:
0.9865282773971558
Epoch 36/60: val loss: 0.047088681114083944 accuracy:
0.9871651785714286
New weights tensor([1.0000, 1.0000, 0.8700, 1.0000, 0.9180, 0.9165,
0.9391, 0.8240, 0.8739])
Learning rate 1.0000000000000002e-06
Epoch 37
avg train loss 30 iterations: 0.040490789183378485 accuracy:
0.9865875244140625
avg train loss 30 iterations: 0.040292286359530395 accuracy:
0.986655056476593
avg train loss 30 iterations: 0.04010472797313944 accuracy:
0.9867218732833862
avg train loss 30 iterations: 0.03990697092792961 accuracy:
0.986793577671051
avg train loss 30 iterations: 0.039705331576990445 accuracy:
0.9868645071983337
Epoch 37/60: val loss: 0.03198653321695539 accuracy: 0.9921875
New weights tensor([1.0000, 1.0000, 0.9406, 1.0000, 0.9426, 0.9382,
0.9141, 0.9112, 0.9400])
Learning rate 1.0000000000000002e-06
Epoch 38
avg train loss 30 iterations: 0.039507347042601094 accuracy:
0.9869314432144165
avg train loss 30 iterations: 0.03931987703770011 accuracy:
0.9870008230209351
avg train loss 30 iterations: 0.039148847482162866 accuracy:
0.9870585203170776
avg train loss 30 iterations: 0.038964921416726486 accuracy:
0.9871211051940918
avg train loss 30 iterations: 0.03877420946484728 accuracy:
0.9871883988380432
Epoch 38/60: val loss: 0.035097300238443756 accuracy:
0.9905133928571429
New weights tensor([1.0000, 1.0000, 0.9107, 1.0000, 0.8683, 0.9736,
0.9701, 0.8282, 0.9434])
Learning rate 1.0000000000000002e-07
```

```
Epoch 39
avg train loss 30 iterations: 0.0385904242635994 accuracy:
0.9872573018074036
avg train loss 30 iterations: 0.038419074044699277 accuracy:
0.9873232245445251
avg train loss 30 iterations: 0.038245078989997454 accuracy:
0.9873831272125244
avg train loss 30 iterations: 0.03806774571598798 accuracy:
0.9874477386474609
avg train loss 30 iterations: 0.03791165778485814 accuracy:
0.9875010848045349
Epoch 39/60: val loss: 0.03591252660927629 accuracy:
0.9854910714285714
New weights tensor([1.0000, 1.0000, 0.8752, 0.9469, 0.8732, 0.8672,
0.9699, 0.8635, 0.8660])
Learning rate 1.0000000000000002e-07
Epoch 40
avg train loss 30 iterations: 0.037746533351137965 accuracy:
0.987550675868988
avg train loss 30 iterations: 0.03757766604447255 accuracy:
0.9876081943511963
avg train loss 30 iterations: 0.037402666761417125 accuracy:
0.987670361995697
avg train loss 30 iterations: 0.03723367601577204 accuracy:
0.98773193359375
avg train loss 30 iterations: 0.037071866839777576 accuracy:
0.9877877235412598
Epoch 40/60: val loss: 0.03229857517119074 accuracy:
0.9910714285714286
New weights tensor([1.0000, 1.0000, 0.9412, 1.0000, 1.0000, 0.9125,
0.9177, 0.7648, 1.0000])
Learning rate 1.0000000000000002e-07
Epoch 41
avg train loss 30 iterations: 0.036930205464300814 accuracy:
0.987834632396698
avg train loss 30 iterations: 0.036809838062666835 accuracy:
0.9878790974617004
avg train loss 30 iterations: 0.03666103639824531 accuracy:
0.9879231452941895
avg train loss 30 iterations: 0.036536131771625555 accuracy:
0.9879616498947144
avg train loss 30 iterations: 0.03638289370409143 accuracy:
0.9880200028419495
Epoch 41/60: val loss: 0.0017691418407983811 accuracy:
0.9994419642857143
New weights tensor([1.0000, 1.0000, 0.9712, 1.0000, 1.0000, 1.0000,
1.0000, 1.0000, 1.0000])
Learning rate 1.0000000000000002e-07
Epoch 42
```

```
avg train loss 30 iterations: 0.0362887254785707 accuracy:
0.988039493560791
avg train loss 30 iterations: 0.0361712361838856 accuracy:
0.9880819320678711
avg train loss 30 iterations: 0.036056238392665314 accuracy:
0.988123893737793
avg train loss 30 iterations: 0.03592358033214249 accuracy:
0.9881704449653625
avg train loss 30 iterations: 0.03582402807841582 accuracy:
0.9882116317749023
Epoch 42/60: val loss: 0.0007248316655282647 accuracy: 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.])
Learning rate 1.0000000000000002e-07
Epoch 43
avg train loss 30 iterations: 0.03572268280854898 accuracy:
0.988239586353302
avg train loss 30 iterations: 0.03559909751422901 accuracy:
0.9882848858833313
avg train loss 30 iterations: 0.03549555992833632 accuracy:
0.9883201122283936
avg train loss 30 iterations: 0.035355382073600485 accuracy:
0.9883695244789124
avg train loss 30 iterations: 0.03522961983360065 accuracy:
0.9884136319160461
Epoch 43/60: val loss: 0.0021153953656331786 accuracy:
0.9994419642857143
New weights tensor([1.0000, 1.0000, 0.9695, 1.0000, 1.0000, 1.0000,
1.0000, 1.0000, 1.0000])
Learning rate 1.0000000000000002e-07
Epoch 44
avg train loss 30 iterations: 0.03510191915284869 accuracy:
0.9884638786315918
avg train loss 30 iterations: 0.03496392996633769 accuracy:
0.9885166883468628
avg train loss 30 iterations: 0.034825135261854094 accuracy:
0.9885643124580383
avg train loss 30 iterations: 0.0346790003834636 accuracy:
0.9886161684989929
avg train loss 30 iterations: 0.03453547124870346 accuracy:
0.9886676073074341
Epoch 44/60: val loss: 0.019145105903070152 accuracy:
0.9938616071428571
New weights tensor([1.0000, 1.0000, 0.9112, 0.9687, 0.9711, 0.9685,
0.9344, 0.9745, 0.9439])
Learning rate 1.0000000000000002e-07
Epoch 45
avg train loss 30 iterations: 0.03439570015015938 accuracy:
0.9887155294418335
avg train loss 30 iterations: 0.03427119052634039 accuracy:
```

```
0.9887520670890808
avg train loss 30 iterations: 0.034149299041248964 accuracy:
0.988792896270752
avg train loss 30 iterations: 0.03402270186321652 accuracy:
0.9888333678245544
avg train loss 30 iterations: 0.033895472869460076 accuracy:
0.988878071308136
Epoch 45/60: val loss: 0.030015487746433273 accuracy:
0.9916294642857143
New weights tensor([1.0000, 1.0000, 0.9161, 0.9693, 0.9406, 0.9397,
0.9707, 0.8825, 0.9403])
Learning rate 1.0000000000000002e-07
Epoch 46
avg train loss 30 iterations: 0.033773108601983556 accuracy:
0.9889194369316101
avg train loss 30 iterations: 0.03363922678646609 accuracy:
0.9889633655548096
avg train loss 30 iterations: 0.03351634783628056 accuracy:
0.9890068769454956
avg train loss 30 iterations: 0.03339625488606031 accuracy:
0.9890500903129578
avg train loss 30 iterations: 0.03328578109145112 accuracy:
0.9890883564949036
Epoch 46/60: val loss: 0.051095477912115585 accuracy:
0.9854910714285714
New weights tensor([1.0000, 1.0000, 0.8199, 1.0000, 0.9722, 0.8915,
0.8825, 0.8492, 0.8453])
Learning rate 1.0000000000000002e-07
Epoch 47
avg train loss 30 iterations: 0.03315146794698966 accuracy:
0.9891368746757507
avg train loss 30 iterations: 0.03302412945543196 accuracy:
0.9891833662986755
avg train loss 30 iterations: 0.032904640139390644 accuracy:
0.9892295002937317
avg train loss 30 iterations: 0.03278486217339263 accuracy:
0.9892663955688477
avg train loss 30 iterations: 0.03266960964982651 accuracy:
0.9893073439598083
Epoch 47/60: val loss: 0.026105290291785162 accuracy:
0.9905133928571429
New weights tensor([1.0000, 1.0000, 0.9400, 0.9714, 0.9423, 0.9378,
0.8885, 0.9116, 0.9107])
Learning rate 1.0000000000000002e-07
Epoch 48
avg train loss 30 iterations: 0.032541547801634145 accuracy:
0.9893494844436646
avg train loss 30 iterations: 0.03241867318034552 accuracy:
0.9893941283226013
```

```
avg train loss 30 iterations: 0.032294707303063185 accuracy:
0.9894384145736694
avg train loss 30 iterations: 0.03217483407085217 accuracy:
0.9894822835922241
avg train loss 30 iterations: 0.03206316727653677 accuracy:
0.9895215034484863
Epoch 48/60: val loss: 0.03472471530845463 accuracy:
0.9893973214285714
New weights tensor([0.9698, 1.0000, 0.8831, 1.0000, 0.8676, 0.9737,
0.9120, 0.8836, 0.9431])
Learning rate 1.0000000000000004e-08
Epoch 49
avg train loss 30 iterations: 0.031943097406983666 accuracy:
0.9895618557929993
avg train loss 30 iterations: 0.03183283480667876 accuracy:
0.989600419998169
avg train loss 30 iterations: 0.03171759741190375 accuracy:
0.9896429777145386
avg train loss 30 iterations: 0.03160546943798282 accuracy:
0.9896808862686157
avg train loss 30 iterations: 0.03148766460457218 accuracy:
0.989722728729248
Epoch 49/60: val loss: 0.03271648438580347 accuracy:
0.9910714285714286
New weights tensor([1.0000, 1.0000, 0.9051, 1.0000, 0.9046, 0.9450,
0.9403, 0.8895, 0.9439])
Learning rate 1.0000000000000004e-08
Epoch 50
avg train loss 30 iterations: 0.03136757768620269 accuracy:
0.9897656440734863
avg train loss 30 iterations: 0.03125492843709625 accuracy:
0.9898067712783813
avg train loss 30 iterations: 0.031151567935679737 accuracy:
0.9898434281349182
avg train loss 30 iterations: 0.03104197624948167 accuracy:
0.9898839592933655
avg train loss 30 iterations: 0.030928511305166734 accuracy:
0.9899241328239441
Epoch 50/60: val loss: 0.0425367119353593 accuracy: 0.9871651785714286

New weights tensor([1.0000, 1.0000, 0.9137, 1.0000, 0.9704, 0.8842,
0.8654, 0.7124, 1.0000])
Learning rate 1.0000000000000004e-08
Epoch 51
avg train loss 30 iterations: 0.030842259235194285 accuracy:
0.9899529814720154
avg train loss 30 iterations: 0.030750187930791404 accuracy:
0.989976167678833
avg train loss 30 iterations: 0.030691663849775924 accuracy:
```

```
0.9899950623512268
avg train loss 30 iterations: 0.030593468293948444 accuracy:
0.9900301694869995
avg train loss 30 iterations: 0.030493067947408907 accuracy:
0.9900689721107483
Epoch 51/60: val loss: 0.0006445418238659581 accuracy: 1.0
New weights tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.])
Learning rate 1.0000000000000004e-08
Epoch 52
avg train loss 30 iterations: 0.03042473626585521 accuracy:
0.9900885820388794
avg train loss 30 iterations: 0.030368535031558556 accuracy:
0.9901108145713806
avg train loss 30 iterations: 0.030294034955363956 accuracy:
0.9901328682899475
avg train loss 30 iterations: 0.030207365872425455 accuracy:
0.9901667237281799
avg train loss 30 iterations: 0.030113029032056725 accuracy:
0.9902002811431885
Epoch 52/60: val loss: 0.0006793224330457244 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: 0.030024679959069486 accuracy:
0.990230917930603
avg train loss 30 iterations: 0.02997371611859266 accuracy:
0.9902521371841431
avg train loss 30 iterations: 0.029877071361130728 accuracy:
0.9902889728546143
avg train loss 30 iterations: 0.02978789907200864 accuracy:
0.9903137683868408
avg train loss 30 iterations: 0.02968434156085072 accuracy:
0.990350067615509
Epoch 53/60: val loss: 0.002991016954443434 accuracy:
0.9988839285714286
New weights tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
1.0000, 0.9720, 0.9705])
Learning rate 1.0000000000000004e-08
Epoch 54
avg train loss 30 iterations: 0.02957865159091185 accuracy:
0.9903873205184937
avg train loss 30 iterations: 0.02948820021309456 accuracy:
0.9904192090034485
avg train loss 30 iterations: 0.02938643472216576 accuracy:
0.9904547333717346
avg train loss 30 iterations: 0.029310321610149653 accuracy:
0.9904783964157104
avg train loss 30 iterations: 0.029219064100492992 accuracy:
0.9905058145523071
```

```
Epoch 54/60: val loss: 0.016857115877168587 accuracy:
0.9955357142857143
New weights tensor([1.0000, 1.0000, 0.9400, 0.9687, 0.9714, 1.0000,
1.0000, 0.9492, 0.9437])
Learning rate 1.0000000000000004e-08
Epoch 55
avg train loss 30 iterations: 0.029120260972947624 accuracy:
0.9905417561531067
avg train loss 30 iterations: 0.029027100645372544 accuracy:
0.9905725121498108
avg train loss 30 iterations: 0.02894602758619842 accuracy:
0.9905992150306702
avg train loss 30 iterations: 0.028854030043284164 accuracy:
0.9906295537948608
avg train loss 30 iterations: 0.028768650923872336 accuracy:
0.9906595945358276
Epoch 55/60: val loss: 0.028859459288599152 accuracy:
0.9910714285714286
New weights tensor([1.0000, 1.0000, 0.9165, 0.9693, 0.9120, 0.9403,
0.9704, 0.8831, 0.9403])
Learning rate 1.0000000000000004e-08
Epoch 56
avg train loss 30 iterations: 0.02868189637791253 accuracy:
0.9906905889511108
avg train loss 30 iterations: 0.02861315554241397 accuracy:
0.9907090663909912
avg train loss 30 iterations: 0.028520269143782404 accuracy:
0.9907422661781311
avg train loss 30 iterations: 0.028439711379513417 accuracy:
0.990767776966095
avg train loss 30 iterations: 0.028347823349930232 accuracy:
0.9908005595207214
Epoch 56/60: val loss: 0.049882836497740106 accuracy:
0.9832589285714286
New weights tensor([1.0000, 1.0000, 0.8199, 0.9664, 0.9447, 0.8910,
0.8543, 0.8232, 0.8453])
Learning rate 1.0000000000000004e-08
Epoch 57
avg train loss 30 iterations: 0.028261511301257095 accuracy:
0.9908304810523987
avg train loss 30 iterations: 0.028173677777156396 accuracy:
0.9908627867698669
avg train loss 30 iterations: 0.028091281447939622 accuracy:
0.9908912181854248
avg train loss 30 iterations: 0.02803286281773801 accuracy:
0.9909157752990723
avg train loss 30 iterations: 0.027956084712685715 accuracy:
0.9909402132034302
Epoch 57/60: val loss: 0.026171099489667022 accuracy:
```

```
0.9910714285714286
New weights tensor([1.0000, 1.0000, 0.9397, 1.0000, 0.9426, 0.9378,
0.8890, 0.9400, 0.8825])
Learning rate 1.0000000000000004e-08
Epoch 58
avg train loss 30 iterations: 0.027863301714152203 accuracy:
0.9909726977348328
avg train loss 30 iterations: 0.02779707556307516 accuracy:
0.9909967184066772
avg train loss 30 iterations: 0.027720973388400433 accuracy:
0.9910241961479187
avg train loss 30 iterations: 0.027646161677522017 accuracy:
0.9910514950752258
avg train loss 30 iterations: 0.027585748141144346 accuracy:
0.9910678863525391
Epoch 58/60: val loss: 0.03487634121743862 accuracy:
0.9888392857142857
New weights tensor([1.0000, 1.0000, 0.9107, 1.0000, 0.8690, 0.9736,
0.9409, 0.8040, 0.9153])
Learning rate 1.0000000000000004e-08
Epoch 59
avg train loss 30 iterations: 0.02750495967262672 accuracy:
0.9910993576049805
avg train loss 30 iterations: 0.027427918033531982 accuracy:
0.9911261200904846
avg train loss 30 iterations: 0.027341810200453967 accuracy:
0.9911562204360962
avg train loss 30 iterations: 0.02726648505812312 accuracy:
0.9911825656890869
avg train loss 30 iterations: 0.027191708960615606 accuracy:
0.9912052154541016
Epoch 59/60: val loss: 0.035277752766757785 accuracy:
0.9899553571428571
New weights tensor([1.0000, 1.0000, 0.9359, 1.0000, 0.9355, 0.9442,
0.9120, 0.8635, 0.8905])
Learning rate 1.0000000000000004e-08
Epoch 60
avg train loss 30 iterations: 0.02710886126767685 accuracy:
0.9912357330322266
avg train loss 30 iterations: 0.027046386882855777 accuracy:
0.9912615418434143
avg train loss 30 iterations: 0.02697450884043586 accuracy:
0.9912872314453125
avg train loss 30 iterations: 0.026903540400547318 accuracy:
0.9913127422332764
avg train loss 30 iterations: 0.026828413012262513 accuracy:
0.9913414716720581
Epoch 60/60: val loss: 0.034503860295704465 accuracy:
0.9899553571428571
New weights tensor([1.0000, 1.0000, 0.8864, 1.0000, 1.0000, 0.9397,
```

```
0.8915, 0.7648, 1.0000])
Learning rate 1.0000000000000004e-08

train done

model.save("best_1")

model.load("best_1")
#model.load("best_model")

Downloading...
From (original): https://drive.google.com/uc?id=1-
3gFYm4uEM4eOg8niDPqdNYICLobtMID
From (redirected): https://drive.google.com/uc?id=1-
3gFYm4uEM4eOg8niDPqdNYICLobtMID&confirm=t&uuid=5e38c17f-7312-4c09-
8832-f124056c1232
To: /content/best_1.pth
100%|██████████| 44.8M/44.8M [00:00<00:00, 59.9MB/s]
<ipython-input-24-8312645e3f37>:20: 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:

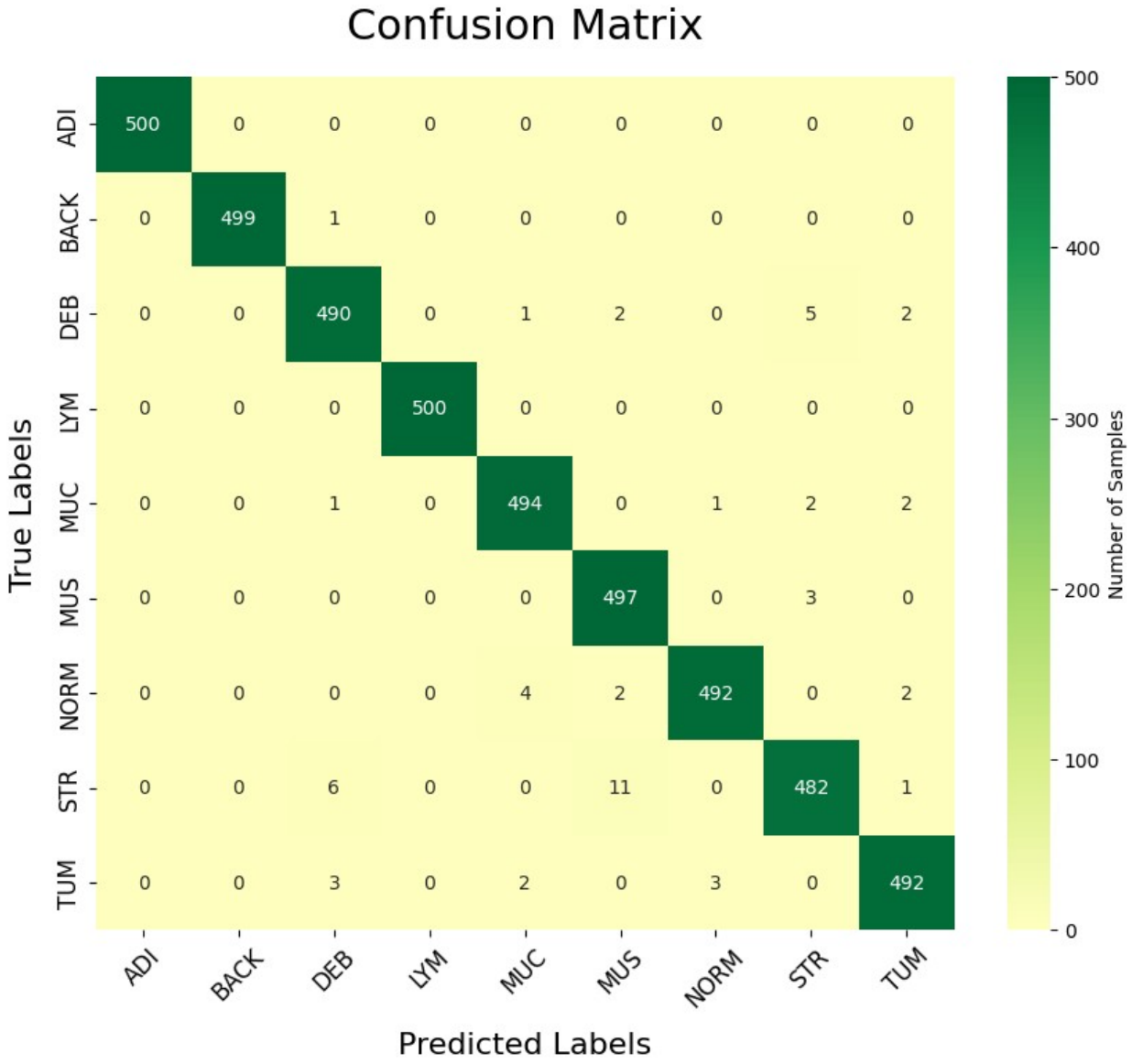
```
# evaluating model on 10% of test dataset
pred_1 = model.test_on_dataset(d_test, limit=0.1)
#confi_matrix(d_test.labels[:len(pred_1)], pred_1, TISSUE_CLASSES)
Metrics.print_all(d_test.labels[:len(pred_1)], pred_1, '10% of test')

{"model_id":"1c96151c3e924303b0d32240b69cfd93","version_major":2,"vers
ion_minor":0}

metrics for 10% of test:
      accuracy 1.0000:
      balanced accuracy 1.0000:
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/
_classification.py:409: UserWarning: A single label was found in
'y_true' and 'y_pred'. For the confusion matrix to have the correct
shape, use the 'labels' parameter to pass all known labels.
  warnings.warn(
```

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":"a5d70fd2e9624057a83253b216cc15f9","version_major":2,"vers
ion_minor":0}
```



Class	True Positive (TP)	False Positive (FP)
ADI	500	0
BACK	499	0
DEB	490	11
LYM	500	0
MUC	494	7
MUS	497	15
NORM	492	4
STR	482	10
TUM	492	7

metrics for test:
accuracy 0.9880:
balanced accuracy 0.9880:

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_model')
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=1L9PiS1JbKKOnxaGleh85wfz_yP5lf0H8
From (redirected): https://drive.google.com/uc?
id=1L9PiS1JbKKOnxaGleh85wfz_yP5lf0H8&confirm=t&uuid=6c5c6619-333c-
44c5-86e1-399aed643f53
To: /content/best_model.pth
100%|██████████| 44.8M/44.8M [00:01<00:00, 34.3MB/s]
<ipython-input-24-8312645e3f37>:20: 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, 126MB/s]

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



{"model_id":"5cee8abaf16e42eca00d831f7786c80b","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()

/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283:
DeprecationWarning: `should_run_async` will not call `transform_cell`
automatically in the future. Please pass the result to
`transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

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 нейроной 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 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')
```