ДОДАТОК А

Вихідний код програми

```
import os
import time
from random import randint

import numpy as np
import pandas as pd
from scipy import stats
from tqdm import tqdm

from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVR
from sklearn.model_selection import KFold

import nibabel as nib
import pydicom as pdm
import nilearn as nl
import nilearn.plotting as nlplt
import h5py

import matplotlib.pyplot as plt
import matplotlib.animation as anim
from matplotlib import cm

import seaborn as sns
import imageio
from skimage.transform import resize
from skimage.util import montage

from IPython.display import Image as show_gif
from IPython.display import clear_output
from IPython.display import YouTubeVideo

import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
```

```
import torch.nn.functional as F

from torch.optim import Adam
from torch.optim.lr_scheduler import ReduceLROnPlateau
from torch.nn import MSELoss, Module, Parameter


import albumentations as A
from albumentations import Compose, HorizontalFlip
from albumentations.pytorch import ToTensor, ToTensorV2

class AdELU(Module):
    def __init__(self, init_r=1.0, init_l=0.1):
        super().__init__()
        self.A_r = Parameter(torch.tensor(init_r, requires_grad=True))
        self.A_l = Parameter(torch.tensor(init_l, requires_grad=True))

    def forward(self, input):
        return torch.where(input < 0, self.A_l * (torch.exp(input) - 1), self.A_r * input)

class AdPReLU(Module):
    def __init__(self, init_r=1.0, init_l=0.1):
        super().__init__()
        self.A_r = Parameter(torch.tensor(init_r, requires_grad=True))
        self.A_l = Parameter(torch.tensor(init_l, requires_grad=True))

    def forward(self, input):
        return torch.where(input <= 0, self.A_l * input, self.A_r * input)

class AdSoftplus(Module):
    def __init__(self, init_r=1.0, init_l=0.1):
        super().__init__()
        self.k = Parameter(torch.tensor(init_r, requires_grad=True))
        self.A_l = Parameter(torch.tensor(init_l, requires_grad=False))

    def forward(self, input):
        return (1/self.k) * torch.log(1 + torch.exp(input * self.k))

class Image3dToGIF3d:
    """
    Displaying 3D images in 3d axes.
    Parameters:
```

```
    img_dim: shape of cube for resizing.
    figsize: figure size for plotting in inches.
"""
def __init__(self,
             img_dim: tuple = (55, 55, 55),
             figsize: tuple = (15, 10),
             binary: bool = False,
             normalizing: bool = True,
            ):
    """Initialization."""
    self.img_dim = img_dim
    print(img_dim)
    self.figsize = figsize
    self.binary = binary
    self.normalizing = normalizing


def _explode(self, data: np.ndarray):
    """
    Takes: array and return an array twice as large in each dimension,
    with an extra space between each voxel.
    """
    shape_arr = np.array(data.shape)
    size = shape_arr[:3] * 2 - 1
    exploded = np.zeros(np.concatenate([size, shape_arr[3:]]),
                        dtype=data.dtype)
    exploded[::2, ::2, ::2] = data
    return exploded


def _expand_coordinates(self, indices: np.ndarray):
    x, y, z = indices
    x[1::2, :, :] += 1
    y[:, 1::2, :] += 1
    z[:, :, 1::2] += 1
    return x, y, z


def _normalize(self, arr: np.ndarray):
    """Normilize image value between 0 and 1."""
    arr_min = np.min(arr)
    return (arr - arr_min) / (np.max(arr) - arr_min)



def _scale_by(self, arr: np.ndarray, factor: int):
```

```
    """
    Scale 3d Image to factor.
    Parameters:
        arr: 3d image for scalling.
        factor: factor for scalling.
    """
    mean = np.mean(arr)
    return (arr - mean) * factor + mean

def get_transformed_data(self, data: np.ndarray):
    """Data transformation: normalization, scaling, resizing."""
    if self.binary:
        resized_data = resize(data, self.img_dim, preserve_range=True)
        return np.clip(resized_data.astype(np.uint8), 0, 1).astype(np.float32)

    norm_data = np.clip(self._normalize(data)-0.1, 0, 1) ** 0.4
    scaled_data = np.clip(self._scale_by(norm_data, 2) - 0.1, 0, 1)
    resized_data = resize(scaled_data, self.img_dim, preserve_range=True)

    return resized_data

def plot_cube(self,
              cube,
              title: str = '',
              init_angle: int = 0,
              make_gif: bool = False,
              path_to_save: str = 'filename.gif'
             ):
    """
    Plot 3d data.
    Parameters:
        cube: 3d data
        title: title for figure.
        init_angle: angle for image plot (from 0-360).
        make_gif: if True create gif from every 5th frames from 3d image plot.
        path_to_save: path to save GIF file.
        """
    if self.binary:
        facecolors = cm.winter(cube)
        print("binary")
    else:
        if self.normalizing:
```

```
            cube = self._normalize(cube)
        facecolors = cm.gist_stern(cube)
        print("not binary")

    facecolors[:,:,:,-1] = cube
    facecolors = self._explode(facecolors)

    filled = facecolors[:,:,:,-1] != 0
    x, y, z = self._expand_coordinates(np.indices(np.array(filled.shape) + 1))

    with plt.style.context("dark_background"):

        fig = plt.figure(figsize=self.figsize)
        ax = fig.gca(projection='3d')

        ax.view_init(30, init_angle)
        ax.set_xlim(right = self.img_dim[0] * 2)
        ax.set_ylim(top = self.img_dim[1] * 2)
        ax.set_zlim(top = self.img_dim[2] * 2)
        ax.set_title(title, fontsize=18, y=1.05)

        ax.voxels(x, y, z, filled, facecolors=facecolors, shade=False)

        if make_gif:
            images = []
            for angle in tqdm(range(0, 360, 5)):
                ax.view_init(30, angle)
                fname = str(angle) + '.png'

                plt.savefig(fname, dpi=120, format='png', bbox_inches='tight')
                images.append(imageio.imread(fname))
                #os.remove(fname)
            imageio.mimsave(path_to_save, images)
            plt.close()

        else:
            plt.show()


class ShowResult:

    def mask_preprocessing(self, mask):
```

```
    """
    Test.
    """
    mask = mask.squeeze().cpu().detach().numpy()
    mask = np.moveaxis(mask, (0, 1, 2, 3), (0, 3, 2, 1))

    mask_WT = np.rot90(montage(mask[0]))
    mask_TC = np.rot90(montage(mask[1]))
    mask_ET = np.rot90(montage(mask[2]))

    return mask_WT, mask_TC, mask_ET

def image_preprocessing(self, image):
    """
    Returns image flair as mask for overlaping gt and predictions.
    """
    image = image.squeeze().cpu().detach().numpy()
    image = np.moveaxis(image, (0, 1, 2, 3), (0, 3, 2, 1))
    flair_img = np.rot90(montage(image[0]))
    return flair_img

def plot(self, image, ground_truth, prediction):
    image = self.image_preprocessing(image)
    gt_mask_WT, gt_mask_TC, gt_mask_ET = self.mask_preprocessing(ground_truth)
    pr_mask_WT, pr_mask_TC, pr_mask_ET = self.mask_preprocessing(prediction)

    fig, axes = plt.subplots(1, 2, figsize = (35, 30))

    [ax.axis("off") for ax in axes]
    axes[0].set_title("Ground Truth", fontsize=35, weight='bold')
    axes[0].imshow(image, cmap ='bone')
    axes[0].imshow(np.ma.masked_where(gt_mask_WT == False, gt_mask_WT),
            cmap='cool_r', alpha=0.6)
    axes[0].imshow(np.ma.masked_where(gt_mask_TC == False, gt_mask_TC),
            cmap='autumn_r', alpha=0.6)
    axes[0].imshow(np.ma.masked_where(gt_mask_ET == False, gt_mask_ET),
            cmap='autumn', alpha=0.6)

    axes[1].set_title("Prediction", fontsize=35, weight='bold')
    axes[1].imshow(image, cmap ='bone')
    axes[1].imshow(np.ma.masked_where(pr_mask_WT == False, pr_mask_WT),
            cmap='cool_r', alpha=0.6)
```

```
        axes[1].imshow(np.ma.masked_where(pr_mask_TC == False, pr_mask_TC),
                cmap='autumn_r', alpha=0.6)
        axes[1].imshow(np.ma.masked_where(pr_mask_ET == False, pr_mask_ET),
                cmap='autumn', alpha=0.6)
        plt.tight_layout()
        plt.show()

def get_all_csv_file(root: str) -> list:
    """Extraction all unique ids from file names."""
    ids = []
    for dirname, _, filenames in os.walk(root):
        for filename in filenames:
            path = os.path.join(dirname, filename)
            if path.endswith(".csv"):
                ids.append(path)
    ids = list(set(filter(None, ids)))
    print(f"Extracted {len(ids)} csv files.")
    return ids
survival_info_df = pd.read_csv('../input/brats20-dataset-training-validation/BraTS2020_TrainingData/MICCAI_BraTS2020_TrainingData/survival_info.csv')
name_mapping_df = pd.read_csv('../input/brats20-dataset-training-validation/BraTS2020_TrainingData/MICCAI_BraTS2020_TrainingData/name_mapping.csv')

name_mapping_df.rename({'BraTS_2020_subject_ID': 'Brats20ID'}, axis=1, inplace=True)

df = survival_info_df.merge(name_mapping_df, on="Brats20ID", how="right")

paths = []
for _, row  in df.iterrows():

    id_ = row['Brats20ID']
    phase = id_.split("_")[-2]

    if phase == 'Training':
        path = os.path.join(config.train_root_dir, id_)
    else:
        path = os.path.join(config.test_root_dir, id_)
    paths.append(path)

df['path'] = paths
train_data = df.loc[df['Age'].notnull()].reset_index(drop=True)
train_data["Age_rank"] =  train_data["Age"] // 10 * 10
```

```
train_data = train_data.loc[train_data['Brats20ID'] != 'BraTS20_Training_355'].reset_index(drop=True, )

skf = StratifiedKFold(
    n_splits=7, random_state=config.seed, shuffle=True
)
for i, (train_index, val_index) in enumerate(
        skf.split(train_data, train_data["Age_rank"])
        ):
        train_data.loc[val_index, "fold"] = i

train_df = train_data.loc[train_data['fold'] != 0].reset_index(drop=True)
val_df = train_data.loc[train_data['fold'] == 0].reset_index(drop=True)

test_df = df.loc[~df['Age'].notnull()].reset_index(drop=True)
train_data.to_csv(config.path_to_csv, index=False)

class BratsDataset(Dataset):
    def __init__(self, df: pd.DataFrame, phase: str="test", is_resize: bool=True):
        self.df = df
        self.phase = phase
        self.augmentations = get_augmentations(phase)
        self.data_types = ['_flair.nii', '_t1.nii', '_t1ce.nii', '_t2.nii']
        self.is_resize = is_resize

    def __len__(self):
        return self.df.shape[0]

    def __getitem__(self, idx):
        id_ = self.df.loc[idx, 'Brats20ID']
        root_path = self.df.loc[self.df['Brats20ID'] == id_]['path'].values[0]
        # load all modalities
        images = []
        for data_type in self.data_types:
            img_path = os.path.join(root_path, id_ + data_type)
            img = self.load_img(img_path)#.transpose(2, 0, 1)

            if self.is_resize:
                img = self.resize(img)

            img = self.normalize(img)
            images.append(img)
        img = np.stack(images)
```

```
    img = np.moveaxis(img, (0, 1, 2, 3), (0, 3, 2, 1))

    if self.phase != "test":
        mask_path =  os.path.join(root_path, id_ + "_seg.nii")
        mask = self.load_img(mask_path)

        if self.is_resize:
            mask = self.resize(mask)

        mask = self.preprocess_mask_labels(mask)

        if self.is_resize:
            mask = np.clip(mask.astype(np.uint8), 0, 1).astype(np.float32)
            mask = np.clip(mask, 0, 1)

        augmented = self.augmentations(image=img.astype(np.float32),
                                       mask=mask.astype(np.float32))

        img = augmented['image']
        mask = augmented['mask']


        return {
            "Id": id_,
            "image": img,
            "mask": mask,
        }

    return {
        "Id": id_,
        "image": img,
    }

def load_img(self, file_path):
    data = nib.load(file_path)
    data = np.asarray(data.dataobj)
    return data

def normalize(self, data: np.ndarray):
    data_min = np.min(data)
    return (data - data_min) / (np.max(data) - data_min)
```

```
    def resize(self, data: np.ndarray):
        data = resize(data, (78, 60, 60), preserve_range=True)
        return data

    def preprocess_mask_labels(self, mask: np.ndarray):
        mask_WT = mask.copy()
        mask_WT[(mask == 1) & (mask == 2) & (mask == 4)] = 1

        mask_TC = mask.copy()
        mask_TC[(mask_TC == 1) & (mask_TC == 4)] = 1
        mask_TC[mask_TC == 2] = 0

        mask_ET = mask.copy()
        mask_ET[(mask_ET == 1) & (mask_ET == 2)] = 0
        mask_ET[mask_ET == 4] = 1

        mask = np.stack([mask_WT, mask_TC, mask_ET])
        mask = np.moveaxis(mask, (0, 1, 2, 3), (0, 3, 2, 1))
        return mask

def get_augmentations(phase):
#     list_transforms = [HorizontalFlip(p=0.5)]
    list_transforms = []

    list_trfms = Compose(list_transforms)
    return list_trfms



def get_dataloader(
    dataset: torch.utils.data.Dataset,
    path_to_csv: str,
    phase: str,
    fold: int = 0,
    batch_size: int = 1,
    num_workers: int = 4,
):
    '''Returns: dataloader for the model training'''
    df = pd.read_csv(config.path_to_csv)

    train_df = df.loc[df['fold'] != fold].reset_index(drop=True)
    val_df = df.loc[df['fold'] == fold].reset_index(drop=True)
```

```
    df = train_df if phase == "train" else val_df
    dataset = dataset(df, phase)
    dataloader = DataLoader(
        dataset,
        batch_size=batch_size,
        num_workers=num_workers,
        pin_memory=True,
        shuffle=True,
    )
    return dataloader

def dice_coef_metric(probabilities: torch.Tensor,
                     truth: torch.Tensor,
                     treshold: float = 0.5,
                     eps: float = 1e-9) -> np.ndarray:
    """
    Calculate Dice score for data batch.
    Params:
        probobilities: model outputs after activation function.
        truth: truth values.
        threshold: threshold for probabilities.
        eps: additive to refine the estimate.
        Returns: dice score aka f1.
    """
    scores = []
    num = probabilities.shape[0]
    predictions = (probabilities >= treshold).float()
    assert(predictions.shape == truth.shape)
    for i in range(num):
        prediction = predictions[i]
        truth_ = truth[i]
        intersection = 2.0 * (truth_ * prediction).sum()
        union = truth_.sum() + prediction.sum()
        if truth_.sum() == 0 and prediction.sum() == 0:
            scores.append(1.0)
        else:
            scores.append((intersection + eps) / union)
    return np.mean(scores)


def jaccard_coef_metric(probabilities: torch.Tensor,
                        truth: torch.Tensor,
```

```
            treshold: float = 0.5,
            eps: float = 1e-9) -> np.ndarray:
    """
    Calculate Jaccard index for data batch.
    Params:
        probobilities: model outputs after activation function.
        truth: truth values.
        threshold: threshold for probabilities.
        eps: additive to refine the estimate.
        Returns: jaccard score aka iou."
    """
    scores = []
    num = probabilities.shape[0]
    predictions = (probabilities >= treshold).float()
    assert(predictions.shape == truth.shape)

    for i in range(num):
        prediction = predictions[i]
        truth_ = truth[i]
        intersection = (prediction * truth_).sum()
        union = (prediction.sum() + truth_.sum()) - intersection + eps
        if truth_.sum() == 0 and prediction.sum() == 0:
            scores.append(1.0)
        else:
            scores.append((intersection + eps) / union)
    return np.mean(scores)

class Meter:
    '''factory for storing and updating iou and dice scores.'''
    def __init__(self, treshold: float = 0.5):
        self.threshold: float = treshold
        self.dice_scores: list = []
        self.iou_scores: list = []

    def update(self, logits: torch.Tensor, targets: torch.Tensor):
        """
        Takes: logits from output model and targets,
        calculates dice and iou scores, and stores them in lists.
        """
        probs = torch.sigmoid(logits)
        dice = dice_coef_metric(probs, targets, self.threshold)
        iou = jaccard_coef_metric(probs, targets, self.threshold)
```

```
        self.dice_scores.append(dice)
        self.iou_scores.append(iou)

    def get_metrics(self) -> np.ndarray:
        """
        Returns: the average of the accumulated dice and iou scores.
        """
        dice = np.mean(self.dice_scores)
        iou = np.mean(self.iou_scores)
        return dice, iou


class DiceLoss(nn.Module):
    """Calculate dice loss."""
    def __init__(self, eps: float = 1e-9):
        super(DiceLoss, self).__init__()
        self.eps = eps

    def forward(self,
                logits: torch.Tensor,
                targets: torch.Tensor) -> torch.Tensor:

        num = targets.size(0)
        probability = torch.sigmoid(logits)
        probability = probability.view(num, -1)
        targets = targets.view(num, -1)
        assert(probability.shape == targets.shape)

        intersection = 2.0 * (probability * targets).sum()
        union = probability.sum() + targets.sum()
        dice_score = (intersection + self.eps) / union
        #print("intersection", intersection, union, dice_score)
        return 1.0 - dice_score


class BCEDiceLoss(nn.Module):
    """Compute objective loss: BCE loss + DICE loss."""
    def __init__(self):
        super(BCEDiceLoss, self).__init__()
        self.bce = nn.BCEWithLogitsLoss()
        self.dice = DiceLoss()
```

```
    def forward(self,
                logits: torch.Tensor,
                targets: torch.Tensor) -> torch.Tensor:
        assert(logits.shape == targets.shape)
        dice_loss = self.dice(logits, targets)
        bce_loss = self.bce(logits, targets)

        return bce_loss + dice_loss

# helper functions for testing.
def dice_coef_metric_per_classes(probabilities: np.ndarray,
                                 truth: np.ndarray,
                                 treshold: float = 0.5,
                                 eps: float = 1e-9,
                                 classes: list = ['WT', 'TC', 'ET']) -> np.ndarray:
    """
    Calculate Dice score for data batch and for each class.
    Params:
        probobilities: model outputs after activation function.
        truth: model targets.
        threshold: threshold for probabilities.
        eps: additive to refine the estimate.
        classes: list with name classes.
        Returns: dict with dice scores for each class.
    """
    scores = {key: list() for key in classes}
    num = probabilities.shape[0]
    num_classes = probabilities.shape[1]
    predictions = (probabilities >= treshold).astype(np.float32)
    assert(predictions.shape == truth.shape)

    for i in range(num):
        for class_ in range(num_classes):
            prediction = predictions[i][class_]
            truth_ = truth[i][class_]
            intersection = 2.0 * (truth_ * prediction).sum()
            union = truth_.sum() + prediction.sum()
            if truth_.sum() == 0 and prediction.sum() == 0:
                scores[classes[class_]].append(1.0)
            else:
                scores[classes[class_]].append((intersection + eps) / union)
```

```
    return scores


def jaccard_coef_metric_per_classes(probabilities: np.ndarray,
                truth: np.ndarray,
                treshold: float = 0.5,
                eps: float = 1e-9,
                classes: list = ['WT', 'TC', 'ET']) -> np.ndarray:
    """
    Calculate Jaccard index for data batch and for each class.
    Params:
        probobilities: model outputs after activation function.
        truth: model targets.
        threshold: threshold for probabilities.
        eps: additive to refine the estimate.
        classes: list with name classes.
        Returns: dict with jaccard scores for each class."
    """
    scores = {key: list() for key in classes}
    num = probabilities.shape[0]
    num_classes = probabilities.shape[1]
    predictions = (probabilities >= treshold).astype(np.float32)
    assert(predictions.shape == truth.shape)

    for i in range(num):
        for class_ in range(num_classes):
            prediction = predictions[i][class_]
            truth_ = truth[i][class_]
            intersection = (prediction * truth_).sum()
            union = (prediction.sum() + truth_.sum()) - intersection + eps
            if truth_.sum() == 0 and prediction.sum() == 0:
                scores[classes[class_]].append(1.0)
            else:
                scores[classes[class_]].append((intersection + eps) / union)

    return scores

    class DoubleConv(nn.Module):
        """(Conv3D -> BN -> ReLU) * 2"""
        def __init__(self, in_channels, out_channels, num_groups=8):
            super().__init__()
```

```
        self.conv1 = nn.Conv3d(in_channels, out_channels, kernel_size=3, stride=1, padding=1)
        self.groupnorm1 = nn.GroupNorm(num_groups=num_groups, num_channels=out_channels)
        self.adelu1 = AdELU()
#         self.adprelu1 = AdPReLU()
#         self.adsoftplus1  = AdSoftplus()
#         self.relu1  = nn.ReLU()
        self.conv2 = nn.Conv3d(out_channels, out_channels, kernel_size=3, stride=1, padding=1)
        self.groupnorm2 = nn.GroupNorm(num_groups=num_groups, num_channels=out_channels)
        self.adelu2 = AdELU()
#         self.adprelu2 = AdPReLU()
#         self.adsoftplus2  = AdSoftplus()
#         self.relu2 = nn.ReLU()

    def forward(self, x):
        x = self.conv1(x)
        x = self.groupnorm1(x)
        x = self.adelu1(x)

        x = self.conv2(x)
        x = self.groupnorm2(x)
        x = self.adelu2(x)
        return x


class Down(nn.Module):

    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.MaxPool3d(2, 2),
            DoubleConv(in_channels, out_channels)
        )
    def forward(self, x):
        return self.encoder(x)


class Up(nn.Module):

    def __init__(self, in_channels, out_channels, trilinear=True):
        super().__init__()

        if trilinear:
```

```
            self.up = nn.Upsample(scale_factor=2, mode='trilinear', align_corners=True)
        else:
            self.up = nn.ConvTranspose3d(in_channels // 2, in_channels // 2, kernel_size=2, stride=2)

        self.decoder = DoubleConv(in_channels, out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)

        diffZ = x2.size()[2] - x1.size()[2]
        diffY = x2.size()[3] - x1.size()[3]
        diffX = x2.size()[4] - x1.size()[4]
        x1 = F.pad(x1, [diffX // 2, diffX - diffX // 2, diffY // 2, diffY - diffY // 2, diffZ // 2, diffZ - diffZ // 2])

        x = torch.cat([x2, x1], dim=1)
        return self.decoder(x)


class Out(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.conv = nn.Conv3d(in_channels, out_channels, kernel_size = 1)

    def forward(self, x):
        return self.conv(x)


class UNet3d(nn.Module):
    def __init__(self, in_channels, n_classes, n_channels):
        super().__init__()
        self.in_channels = in_channels
        self.n_classes = n_classes
        self.n_channels = n_channels

        self.input = DoubleConv(in_channels, n_channels)
        self.encoder1 = Down(n_channels, 2 * n_channels)
        self.encoder2 = Down(2 * n_channels, 4 * n_channels)
        self.encoder3 = Down(4 * n_channels, 8 * n_channels)
        self.encoder4 = Down(8 * n_channels, 8 * n_channels)

        self.decoder1 = Up(16 * n_channels, 4 * n_channels)
        self.decoder2 = Up(8 * n_channels, 2 * n_channels)
```

```
        self.decoder3 = Up(4 * n_channels, n_channels)
        self.decoder4 = Up(2 * n_channels, n_channels)
        self.out = Out(n_channels, n_classes)

    def forward(self, x):
        x1 = self.input(x)
        x2 = self.encoder1(x1)
        x3 = self.encoder2(x2)
        x4 = self.encoder3(x3)
        x5 = self.encoder4(x4)

        mask = self.decoder1(x5, x4)
        mask = self.decoder2(mask, x3)
        mask = self.decoder3(mask, x2)
        mask = self.decoder4(mask, x1)
        mask = self.out(mask)
        return mask

class Trainer:
    """
    Factory for training proccess.
    Args:
        display_plot: if True - plot train history after each epoch.
        net: neural network for mask prediction.
        criterion: factory for calculating objective loss.
        optimizer: optimizer for weights updating.
        phases: list with train and validation phases.
        dataloaders: dict with data loaders for train and val phases.
        path_to_csv: path to csv file.
        meter: factory for storing and updating metrics.
        batch_size: data batch size for one step weights updating.
        num_epochs: num weights updation for all data.
        accumulation_steps: the number of steps after which the optimization step can be taken
                (https://www.kaggle.com/c/understanding_cloud_organization/discussion/105614).
        lr: learning rate for optimizer.
        scheduler: scheduler for control learning rate.
        losses: dict for storing lists with losses for each phase.
        jaccard_scores: dict for storing lists with jaccard scores for each phase.
        dice_scores: dict for storing lists with dice scores for each phase.
    """
    def __init__(self,
                 net: nn.Module,
```

```
        model_name: str,
        dataset: torch.utils.data.Dataset,
        criterion: nn.Module,
        lr: float,
        accumulation_steps: int,
        batch_size: int,
        fold: int,
        num_epochs: int,
        path_to_csv: str,
        display_plot: bool = True,
       ):

"""Initialization."""
self.device = 'cuda' if torch.cuda.is_available() else 'cpu'
print("device:", self.device)
self.display_plot = display_plot
self.net = net
self.net = self.net.to(self.device)
self.model_name = model_name
self.criterion = criterion
self.optimizer = Adam(self.net.parameters(), lr=lr)
self.scheduler = ReduceLROnPlateau(self.optimizer, mode="min",
                                   patience=2, verbose=True)
self.accumulation_steps = accumulation_steps // batch_size
self.phases = ["train", "val"]
self.num_epochs = num_epochs

self.dataloaders = {
    phase: get_dataloader(
        dataset = dataset,
        path_to_csv = path_to_csv,
        phase = phase,
        fold = fold,
        batch_size = batch_size,
        num_workers = 6
    )
    for phase in self.phases
}
self.best_loss = float("inf")
self.losses = {phase: [] for phase in self.phases}
self.dice_scores = {phase: [] for phase in self.phases}
self.jaccard_scores = {phase: [] for phase in self.phases}
```

```
    def _compute_loss_and_outputs(self,
                                  images: torch.Tensor,
                                  targets: torch.Tensor):
        images = images.to(self.device)
        targets = targets.to(self.device)
        logits = self.net(images)
        loss = self.criterion(logits, targets)
        return loss, logits

    def _do_epoch(self, epoch: int, phase: str):
        print(f"{phase} epoch: {epoch} | time: {time.strftime('%H:%M:%S')}")

        self.net.train() if phase == "train" else self.net.eval()
        meter = Meter()
        dataloader = self.dataloaders[phase]
        total_batches = len(dataloader)
        running_loss = 0.0
        self.optimizer.zero_grad()
        for itr, data_batch in enumerate(tqdm(dataloader)):

#           if phase != 'train' and epoch in [0, 4, 9] and itr == 0:
#               visualize_activations(self.net, data_batch, epoch, device=self.device)

            images, targets = data_batch['image'], data_batch['mask']
            loss, logits = self._compute_loss_and_outputs(images, targets)
            loss = loss / self.accumulation_steps



            if phase == "train":
                loss.backward()
                if (itr + 1) % self.accumulation_steps == 0:
                    self.optimizer.step()
                    self.optimizer.zero_grad()

            running_loss += loss.item()
            meter.update(logits.detach().cpu(),
                         targets.detach().cpu()
                        )

        if (itr + 1) % self.accumulation_steps != 0 and phase == 'train':
            self.optimizer.step()
```

```
#         if phase == 'train' and epoch in [4, 9]:
#             visualize_gradients(self.net, epoch, device=self.device)

        epoch_loss = (running_loss * self.accumulation_steps) / total_batches
        epoch_dice, epoch_iou = meter.get_metrics()

        self.losses[phase].append(epoch_loss)
        self.dice_scores[phase].append(epoch_dice)
        self.jaccard_scores[phase].append(epoch_iou)

        return epoch_loss

    def run(self):
        for epoch in range(self.num_epochs):
            _ = self._do_epoch(epoch, "train")
            with torch.no_grad():
                val_loss = self._do_epoch(epoch, "val")
                self.scheduler.step(val_loss)
            if self.display_plot:
                self._plot_train_history()
                self._save_train_history
            if val_loss < self.best_loss:
                print(f"\n{'#'*20}\nSaved new checkpoint\n{'#'*20}\n")
                self.best_loss = val_loss
                torch.save(self.net.state_dict(), "best_model.pth")
            print()
        self._save_train_history()

    def _plot_train_history(self):
        data = [self.losses, self.dice_scores, self.jaccard_scores]
        colors = ['deepskyblue', "crimson"]
        labels = [
            f"""
            train loss {self.losses['train'][-1]}
            val loss {self.losses['val'][-1]}
            """,

            f"""
            train dice score {self.dice_scores['train'][-1]}
            val dice score {self.dice_scores['val'][-1]}
            """,
```

```
            f"""
            train jaccard score {self.jaccard_scores['train'][-1]}
            val jaccard score {self.jaccard_scores['val'][-1]}
            """,
        ]

        clear_output(True)
        with plt.style.context("seaborn-dark-palette"):
            fig, axes = plt.subplots(3, 1, figsize=(12, 14))
            for i, ax in enumerate(axes):
                ax.plot(data[i]['val'], c=colors[0], label="val")
                ax.plot(data[i]['train'], c=colors[-1], label="train")
                ax.set_title(labels[i])
                ax.legend(loc="upper right")

            plt.tight_layout()
            plt.show()
            plt.savefig(f'./train_history_{self.model_name}.svg', format='svg')

    def load_predtrain_model(self,
                             state_path: str):
        self.net.load_state_dict(torch.load(state_path))
        print("Predtrain model loaded")

    def _save_train_history(self):
        """writing model weights and training logs to files."""
        torch.save(self.net.state_dict(),
                   f"last_epoch_model.pth")

        logs_ = [self.losses, self.dice_scores, self.jaccard_scores]
        log_names_ = ["_loss", "_dice", "_jaccard"]
        logs = [logs_[i][key] for i in list(range(len(logs_)))
                              for key in logs_[i]]
        log_names = [key+log_names_[i]
                     for i in list(range(len(logs_)))
                     for key in logs_[i]
                    ]
        pd.DataFrame(
            dict(zip(log_names, logs))
        ).to_csv(f"train_log_model_{self.model_name}.csv", index=False)
```

```
model = UNet3d(in_channels=4, n_classes=3, n_channels=16).to('cuda')
trainer = Trainer(net=model,
                  model_name='AdELU',
                  dataset=BratsDataset,
                  criterion=BCEDiceLoss(),
                  lr=5e-4,
                  accumulation_steps=8,
                  batch_size=2,
                  fold=0,
                  num_epochs=10,
                  path_to_csv = config.path_to_csv)
```

ДОДАТОК Б

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	_95_ с.
	Інші документи	
2.	Презентаційні матеріали	_12_ плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Скорік В.А.			Адаптивні активаційні функції глибинних нейронних мереж для сегментації тривимірних медичних зображень	Шифр групи	Код напр./спец.
Перевірив		Бодянський Є.В.				СШІм-20-2	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Філатов В.О.					