ДОДАТОК А

Результати моделей

Рисунок А.1 – Графік втрат мережі LeNetCustom на тренувальних та валідаційних даних.

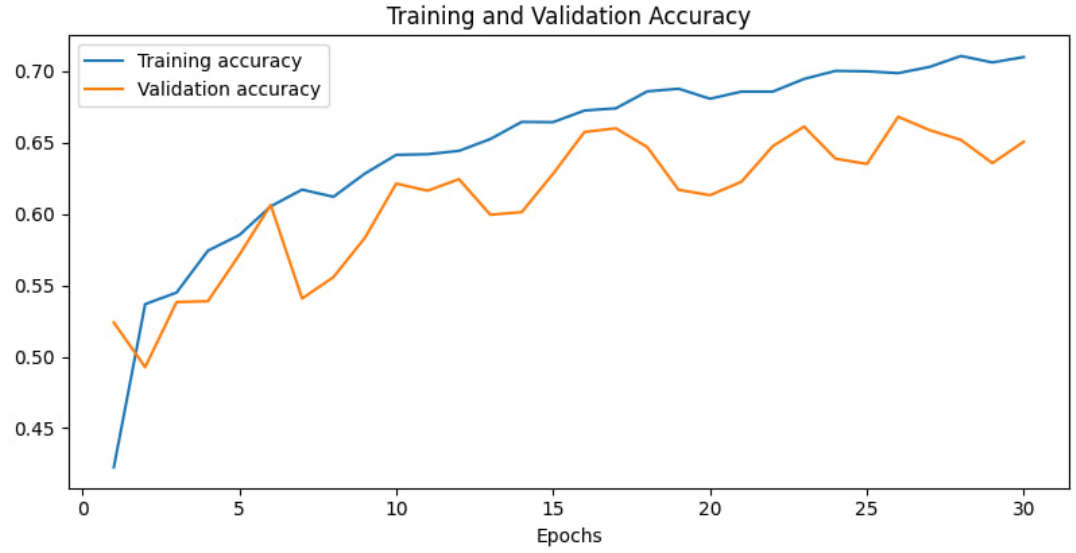

Рисунок А.2 – Графік точності класифікації мережі LeNetCustom на тренувальних та валідаційних даних

Рисунок А.3 – Графік втрат мережі LeNet на тренувальних та валідаційних даних.

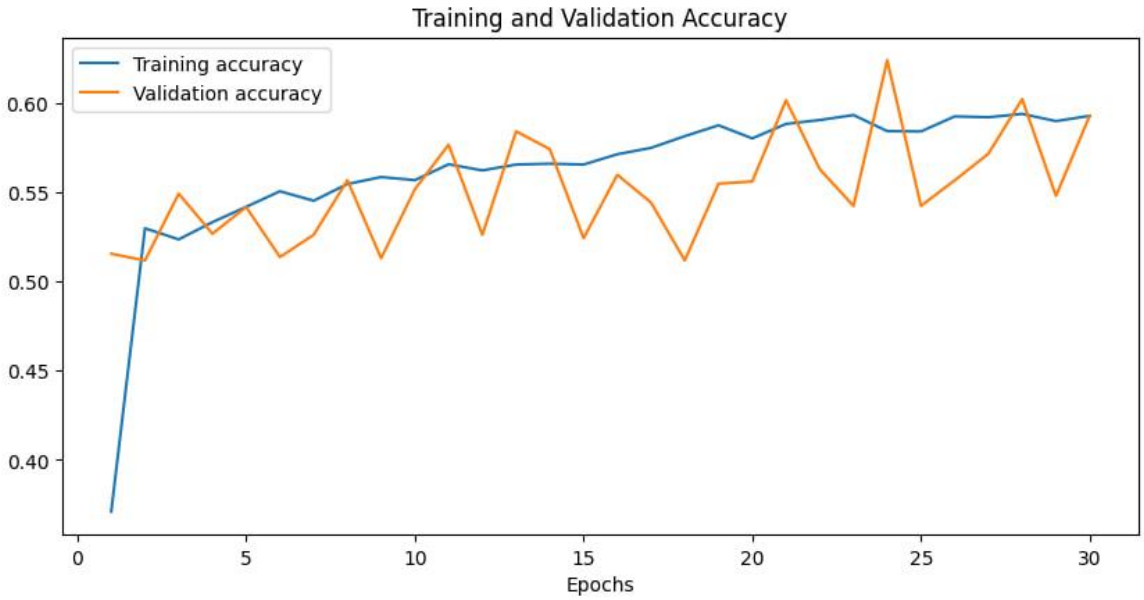

Рисунок А4 – Графік точності класифікації мережі LeNet на тренувальних та валідаційних даних

Рисунок А.5 – Графік втрат мережі ResNet18 на тренувальних та валідаційних даних.

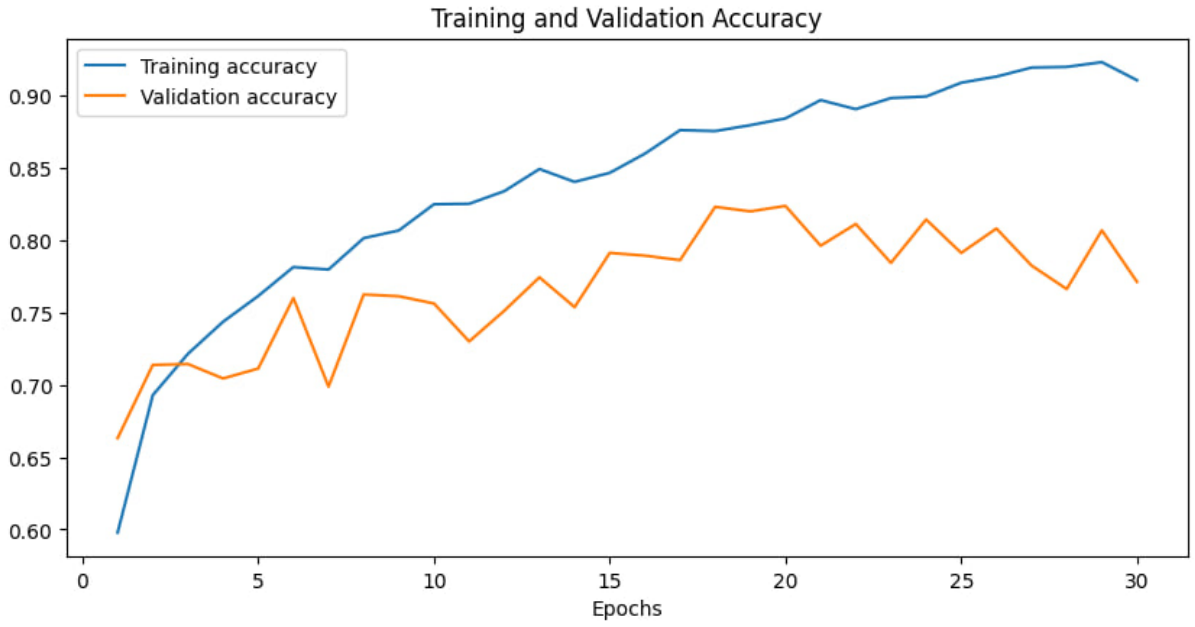

Рисунок А6 – Графік точності класифікації мережі ResNet18 на тренувальних та валідаційних даних

Рисунок А.7 – Графік втрат мережі ResNet152 на тренувальних та валідаційних даних.

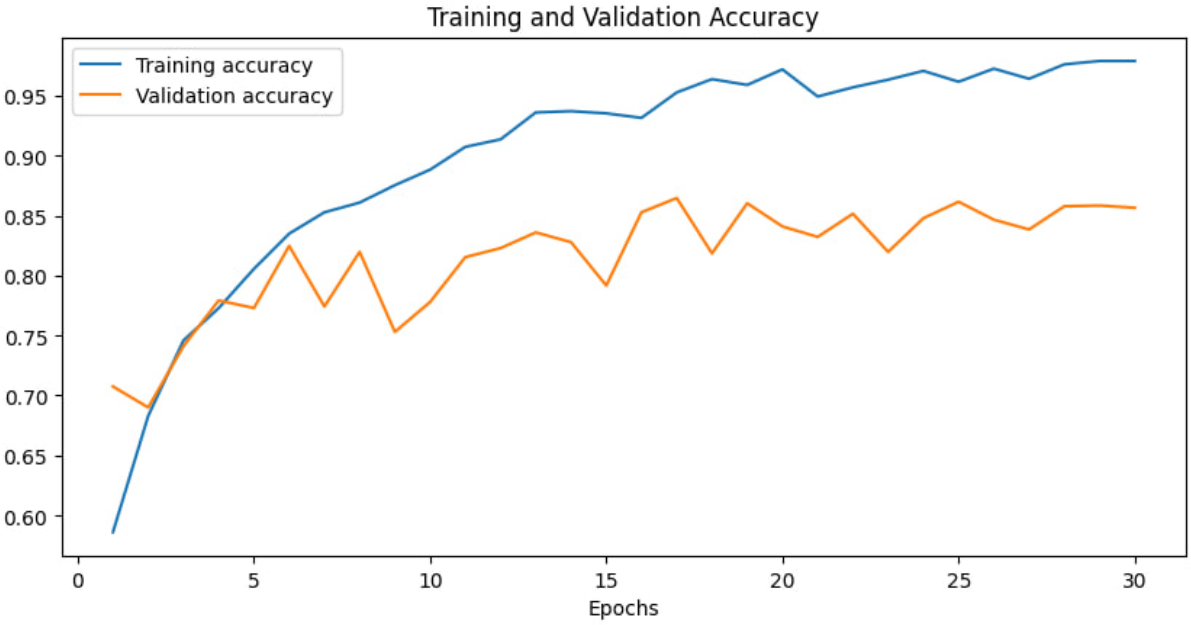

Рисунок А.8 – Графік точності класифікації мережі ResNet152 на тренувальних та валідаційних даних

Рисунок А.9 – Графік втрат мережі VGG19 на тренувальних та валідаційних даних.

Рисунок А.10 – Графік точності класифікації мережі VGG19 на тренувальних та валідаційних даних

Рисунок А.11 – Графік втрат мережі AlexNet на тренувальних та валідаційних даних.

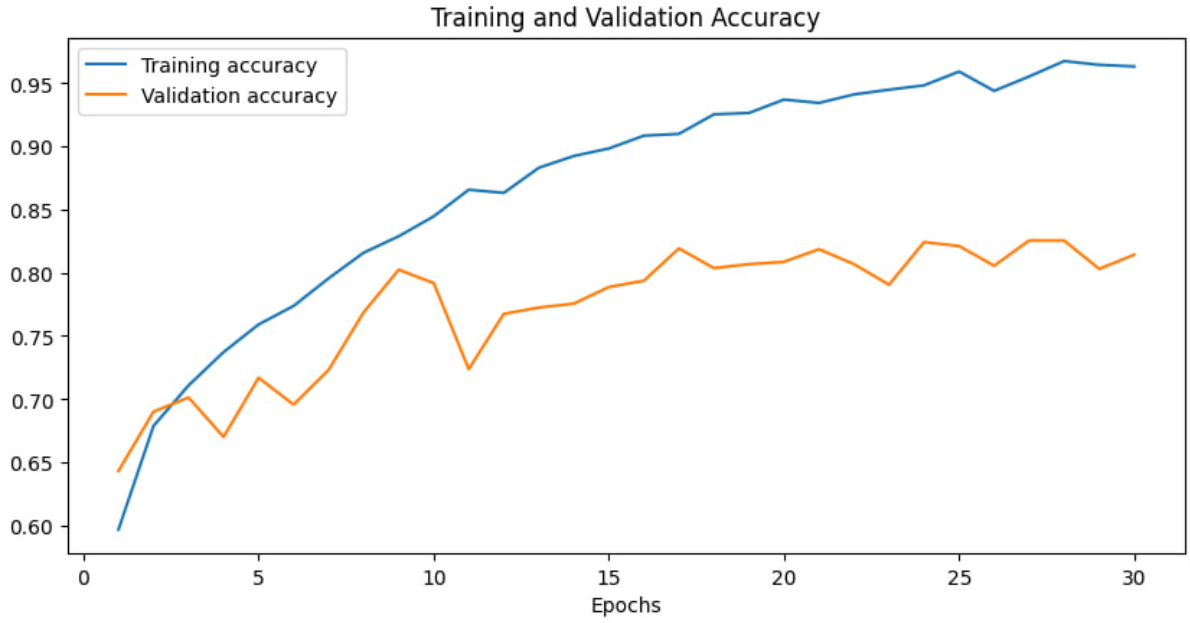

Рисунок А.12 – Графік точності класифікації мережі AlexNet на тренувальних та валідаційних даних

Рисунок А.13 – Графік втрат мережі GoogLeNet на тренувальних та валідаційних даних.

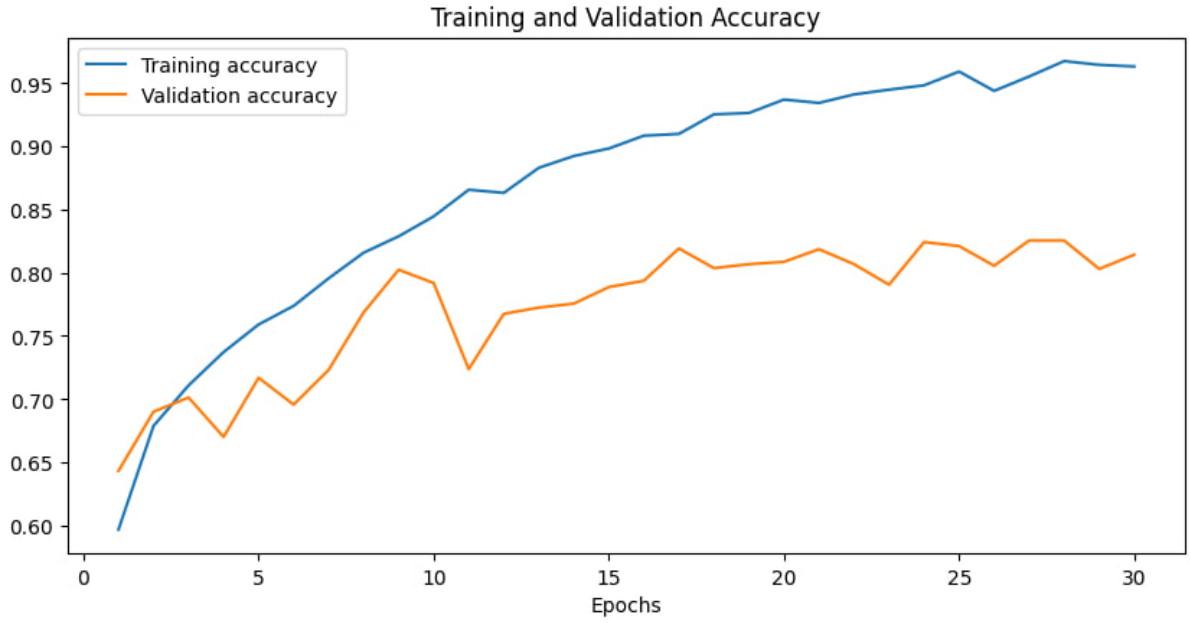

Рисунок А.14 – Графік точності класифікації мережі GoogLeNet на тренувальних та валідаційних даних

Рисунок А.15 – Графік втрат мережі RegNet_Y_32GF на тренувальних та валідаційних даних.

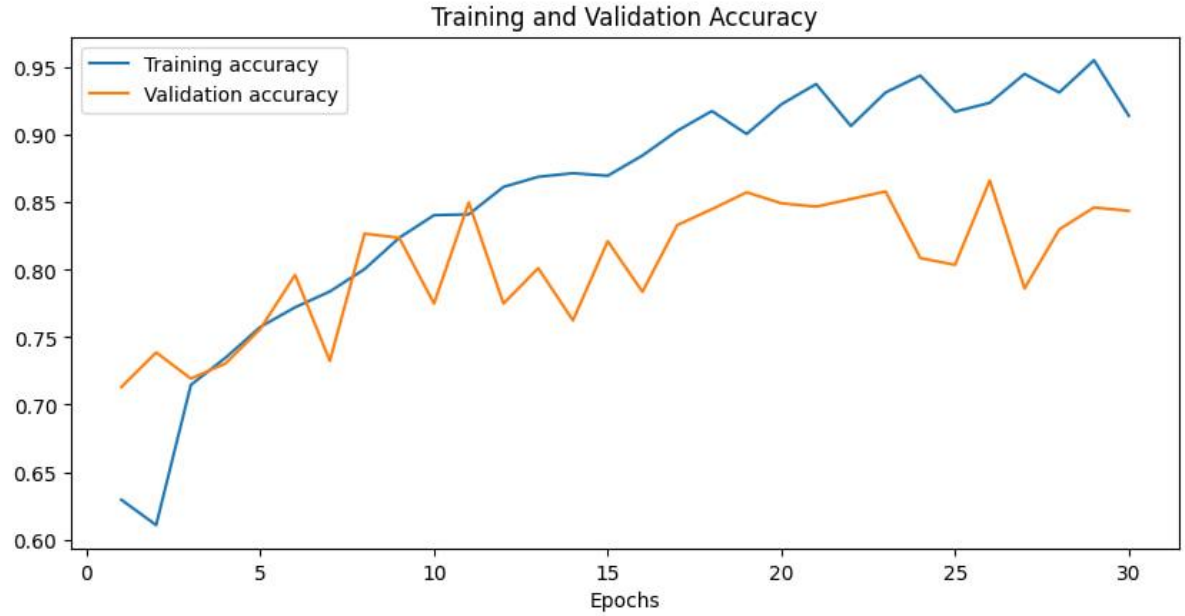

Рисунок А.16 – Графік точності класифікації мережі RegNet_Y_32GF на тренувальних та валідаційних даних

ДОДАТОК Б

Код програми

```
import os
import torch
import imageio
import numpy as np
import pandas as pd
import torchvision
from matplotlib import pyplot as plt
from sklearn.preprocessing import LabelEncoder
import seaborn as sns
import scipy.stats as stats
from torch import nn

#1--------------------------------------------------

data_dir = os.getcwd() + "/HAM/HAM10000"
metadata = pd.read_csv(os.getcwd() + '/HAM/HAM10000_metadata.csv')

# label encoding the seven classes for skin cancers
le = LabelEncoder()
le.fit(metadata['dx'])
LabelEncoder()
#print("Classes:", list(le.classes_))

metadata['label'] = le.transform(metadata["dx"])
metadata.sample(10)



#Визуализация
# Getting a sense of what the distribution of each column looks like
'''
fig = plt.figure(figsize=(15,10))

ax1 = fig.add_subplot(221)
metadata['dx'].value_counts().plot(kind='bar', ax=ax1)
ax1.set_ylabel('Count')
ax1.set_title('Cell Type');

ax2 = fig.add_subplot(222)
```

```
metadata['sex'].value_counts().plot(kind='bar', ax=ax2)
ax2.set_ylabel('Count', size=15)
ax2.set_title('Sex');

ax3 = fig.add_subplot(223)
metadata['localization'].value_counts().plot(kind='bar')
ax3.set_ylabel('Count',size=12)
ax3.set_title('Localization')


ax4 = fig.add_subplot(224)
sample_age = metadata[pd.notnull(metadata['age'])]
sns.distplot(sample_age['age'], fit=stats.norm, color='red');
ax4.set_title('Age')

plt.tight_layout()
plt.show()
'''

#2----------------------------------------------------
#Data Loading and Pre-processing

import os
import shutil

dest_dir = "C:/Users/atom5/PycharmProjects/D1/HAM/HAM10K/"

if not os.path.exists(dest_dir):
    # A path to the folder which has all the images:
    data_dir = os.getcwd() + "/HAM/HAM10000"

    # A path to the folder where you want to store the rearranged images:


    # Read the metadata file:
    metadata = pd.read_csv(os.getcwd() + '/HAM/HAM10000_metadata.csv')
    label = ['bkl', 'nv', 'df', 'mel', 'vasc', 'bcc', 'akiec']
    label_images = []

    # Copy the images into new folder structure:
    for i in label:
        os.mkdir(dest_dir + str(i) + "/")
        sample = metadata[metadata['dx'] == i]['image_id']
```

```
        label_images.extend(sample)
        for id in label_images:
            shutil.copyfile((data_dir + "/" + id + ".jpg"), (dest_dir + i +
"/" + id + ".jpg"))
        label_images = []

#3----------------------
#Overcome Class Imbalance: Median Frequency Balancing

label = [ 'akiec', 'bcc','bkl','df','mel', 'nv',  'vasc']

def estimate_weights_mfb(label):
    import warnings

    warnings.filterwarnings("ignore", category=FutureWarning)

    class_weights = np.zeros_like(label, dtype=np.float64)
    counts = np.zeros_like(label)
    for i, l in enumerate(label):
        counts[i]           =           metadata[metadata['dx']           ==
str(l)]['dx'].value_counts()[0]
    counts = counts.astype(np.float64)
    median_freq = np.median(counts)
    for i, l in enumerate(label):
        class_weights[i] = median_freq / counts[i]
    return class_weights

classweight = estimate_weights_mfb(label)
# for i in range(len(label)):
#     print(label[i], ":", classweight[i])


#4--------------------------------------
#Data Visualization


label = ['akiec', 'bcc', 'bkl', 'df', 'mel', 'nv', 'vasc']
label_images = []
classes = ['actinic keratoses', 'basal cell carcinoma', 'benign keratosis-
like lesions',
           'dermatofibroma',  'melanoma',  'melanocytic  nevi',  'vascular
lesions']
```

```
'''
fig = plt.figure(figsize=(20, 20))
k = range(7)

for i in label:
    sample = metadata[metadata['dx'] == i]['image_id'][:5]
    label_images.extend(sample)

for position, ID in enumerate(label_images):
    labl = metadata[metadata['image_id'] == ID]['dx']
    im_sample = data_dir + "/" + f'/{ID}.jpg'
    im_sample = imageio.imread(im_sample)

    plt.subplot(7, 5, position + 1)
    plt.imshow(im_sample)
    plt.axis('off')

    if position % 5 == 0:
        title = int(position / 5)
        plt.title(classes[title], loc='left', size=20)

plt.tight_layout()
plt.show()
'''

#5----------------------
#Data Augmentation
import torchvision.transforms as transforms

data_dir = os.getcwd() + "/HAM/HAM10000"

# normalization values for pretrained resnet on Imagenet
norm_mean = (0.4914, 0.4822, 0.4465)
norm_std = (0.2023, 0.1994, 0.2010)

batch_size = 10
validation_batch_size = 10

# We compute the weights of individual classes and convert them to tensors
class_weights = estimate_weights_mfb(label)
class_weights = torch.FloatTensor(class_weights)

transform_train = transforms.Compose([
```

```
                    transforms.Resize((224,224)),
                    transforms.RandomHorizontalFlip(),
                    transforms.RandomRotation(degrees=60),
                    transforms.ToTensor(),
                    transforms.Normalize(norm_mean, norm_std),
                    ])

transform_test = transforms.Compose([
                    transforms.Resize((224,224)),
                    transforms.ToTensor(),
                    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023,
0.1994, 0.2010)),
                    ])

#6----------------------------------
#Train, Test and Validation Splits
test_size = 0.2
val_size = 0.2

class Sampler(object):
    """Base class for all Samplers.
    """

    def __init__(self, data_source):
        pass

    def __iter__(self):
        raise NotImplementedError

    def __len__(self):
        raise NotImplementedError

class StratifiedSampler(Sampler):
    """Stratified Sampling
    Provides equal representation of target classes
    """

    def __init__(self, class_vector, test_size=0.2):
        """
        Arguments
        ---------
        class_vector : torch tensor
            a vector of class labels
```

```
        test_size : float
            the proportion of the dataset to include in the test split
        """
        self.n_splits = 1
        self.class_vector = class_vector
        self.test_size = test_size

    def gen_sample_array(self):
        try:
            from sklearn.model_selection import StratifiedShuffleSplit
        except ImportError:
            print('Need scikit-learn for this functionality')
            return [], []  # Return empty indices if scikit-learn is not
available
        import numpy as np

        s          =          StratifiedShuffleSplit(n_splits=self.n_splits,
test_size=self.test_size)
        X = torch.randn(self.class_vector.size(0), 2).numpy()
        y = self.class_vector.numpy()
        s.get_n_splits(X, y)

        train_index, test_index = next(s.split(X, y))
        return train_index, test_index

    def __iter__(self):
        return iter(self.gen_sample_array())

    def __len__(self):
        return len(self.class_vector)

data_dir = 'C:/Users/atom5\PycharmProjects\D1\HAM\HAM10K'

dataset = torchvision.datasets.ImageFolder(root=data_dir)
data_label = [s[1] for s in dataset.samples]

ss = StratifiedSampler(torch.FloatTensor(data_label), test_size)
pre_train_indices, test_indices = ss.gen_sample_array()
# The "pre" is necessary to use array to identify train/ val indices with
indices generated by second sampler

train_label = np.delete(data_label, test_indices, None)
ss = StratifiedSampler(torch.FloatTensor(train_label), test_size)
```

```
train_indices, val_indices = ss.gen_sample_array()
indices = {'train': pre_train_indices[train_indices],  # Indices of second
sampler are used on pre_train_indices
           'val':  pre_train_indices[val_indices],    #  Indices  of  second
sampler are used on pre_train_indices
           'test': test_indices
           }

train_indices = indices['train']
val_indices = indices['val']
test_indices = indices['test']
# print("Train Data Size:", len(train_indices))
# print("Test Data Size:", len(test_indices))
# print("Validation Data Size:", len(val_indices))

#Now we use the Pytorch data loader to load the dataset into the memory.
if __name__ == '__main__':
    SubsetRandomSampler = torch.utils.data.sampler.SubsetRandomSampler

    dataset        =        torchvision.datasets.ImageFolder(root=data_dir,
transform=transform_train)

    train_samples = SubsetRandomSampler(train_indices)
    val_samples = SubsetRandomSampler(val_indices)
    test_samples = SubsetRandomSampler(test_indices)

    train_data_loader         =         torch.utils.data.DataLoader(dataset,
batch_size=batch_size,
                                                    shuffle=False,
num_workers=1, sampler=train_samples)
    validation_data_loader      =      torch.utils.data.DataLoader(dataset,
batch_size=validation_batch_size,
                                                         shuffle=False,
sampler=val_samples)

    dataset        =        torchvision.datasets.ImageFolder(root=data_dir,
transform=transform_test)
    test_data_loader         =         torch.utils.data.DataLoader(dataset,
batch_size=validation_batch_size,
                                                   shuffle=False,
sampler=test_samples)

    # functions to show an image
```

```
    fig = plt.figure(figsize=(10, 15))


    def imshow(img):
        img = img / 2 + 0.5  # denormalize
        npimg = img.numpy()
        npimg = np.clip(npimg, 0, 1)  # Clip values to the valid range [0,
1]
        plt.imshow(np.transpose(npimg, (1, 2, 0)))
        plt.show()


    # get some random training images
    for images, labels in train_data_loader:
        # show images
        imshow(torchvision.utils.make_grid(images))
        # print labels
        print('   '.join('%5s,   '   %   classes[labels[j]]   for   j   in
range(len(labels))))
        break  # Only process the first batch, remove this line if you want
to process all batches

    #plt.show()

    #7-----------------------------------
    #Define a Convolutional Neural Network

    num_classes = len(classes)

    import torch
    import torch.nn as nn
    import torch.nn.functional as F
    class LeNet(nn.Module):
        def __init__(self):
            super(LeNet, self).__init__()
            self.conv1 = nn.Conv2d(3, 6, (5,5), padding=2)
            self.conv2 = nn.Conv2d(6, 16, (5,5))
            self.fc1   = nn.Linear(16*54*54, 120)
            self.fc2   = nn.Linear(120, 84)
            self.fc3   = nn.Linear(84, num_classes)
        def forward(self, x):
            x = F.max_pool2d(F.relu(self.conv1(x)), (2,2))
            x = F.max_pool2d(F.relu(self.conv2(x)), (2,2))
```

```
            x = x.view(-1, self.num_flat_features(x))
            x = F.relu(self.fc1(x))
            x = F.relu(self.fc2(x))
            x = self.fc3(x)
            return x
        def num_flat_features(self, x):
            size = x.size()[1:]
            num_features = 1
            for s in size:
                num_features *= s
            return num_features

    class LeNetCustom(nn.Module):
        def __init__(self):
            super(LeNetCustom, self).__init__()
            self.conv1 = nn.Conv2d(3, 6, (5, 5), padding=2)
            self.conv2 = nn.Conv2d(6, 16, (5, 5))
            self.fc1 = nn.Linear(16 * 54 * 54, 120)
            self.fc2 = nn.Linear(120, 84)
            self.fc3 = nn.Linear(84, num_classes)
            self.dropout = nn.Dropout(0.5)  # Додано Dropout шар
            self.bn1 = nn.BatchNorm2d(6)  # Додано Batch Normalization після
першого згорткового шару
            self.bn2 = nn.BatchNorm2d(16)  # Додано Batch Normalization після
другого згорткового шару

        def forward(self, x):
            x = self.bn1(F.max_pool2d(F.leaky_relu(self.conv1(x)),
                                      (2, 2)))  # Змінено ReLU на Leaky ReLU
та додано Batch Normalization
            x = self.bn2(F.max_pool2d(F.leaky_relu(self.conv2(x)),
                                      (2, 2)))  # Змінено ReLU на Leaky ReLU
та додано Batch Normalization
            x = x.view(-1, self.num_flat_features(x))
            x = F.leaky_relu(self.fc1(x))  # Змінено ReLU на Leaky ReLU
            x = self.dropout(x)  # Додано Dropout
            x = F.leaky_relu(self.fc2(x))  # Змінено ReLU на Leaky ReLU
            x = self.fc3(x)
            return x

        def num_flat_features(self, x):
            size = x.size()[1:]
            num_features = 1
```

```
            for s in size:
                num_features *= s
            return num_features

    # 7.1----------------------------------
    # Define a Loss function and Optimizer
    import torch.optim as optim
    import torch.nn as nn
    import torchvision
    import torchvision.models as models
    from torchvision.models import ResNet18_Weights
    from torchvision.models import ResNet152_Weights
    from torchvision.models import VGG19_Weights
    from torchvision.models import AlexNet_Weights
    from torchvision.models import GoogLeNet_Weights
    from torchvision.models import EfficientNet_B7_Weights
    from torchvision.models import RegNet_Y_128GF_Weights
    from torchvision.models import RegNet_Y_32GF_Weights

    def get_pretrained_model(model_name, total_num_classes):
        if model_name == 'LeNetCustom':
            net = LeNetCustom()
        elif model_name == 'LeNet':
            net = LeNet()
        elif model_name == 'ResNet18':
            net = models.resnet18(weights=ResNet18_Weights.IMAGENET1K_V1)
            net.fc = nn.Linear(net.fc.in_features, total_num_classes)
        elif model_name == 'ResNet152':
            net = models.resnet152(weights=ResNet152_Weights.IMAGENET1K_V2)
            net.fc = nn.Linear(net.fc.in_features, total_num_classes)
        elif model_name == 'VGG19':
            net = models.vgg19(weights=VGG19_Weights.IMAGENET1K_V1)
            net.classifier[6]   =   nn.Linear(net.classifier[6].in_features,
total_num_classes)
        elif model_name == 'AlexNet':
            net = models.alexnet(weights=AlexNet_Weights.IMAGENET1K_V1)
            net.classifier[6]   =   nn.Linear(net.classifier[6].in_features,
total_num_classes)
        elif model_name == 'GoogLeNet':
            net = models.googlenet(weights=GoogLeNet_Weights.IMAGENET1K_V1)
        elif model_name == 'RegNet_Y_32GF':
```

```
            net                                                    =
models.regnet_y_32gf(weights=RegNet_Y_32GF_Weights.IMAGENET1K_SWAG_LINEAR_V
1)
            net.fc = nn.Linear(net.fc.in_features, total_num_classes)
        else:
            raise ValueError("Unknown model name")

        print('Selected model:', model_name)

        return net


    net = get_pretrained_model('LeNetCustom', num_classes)
    # net = get_pretrained_model('LeNet', num_classes)
    # net = get_pretrained_model('ResNet18', num_classes)
    # net = get_pretrained_model('ResNet152', num_classes)
    # net = get_pretrained_model('VGG19', num_classes)
    # net = get_pretrained_model('AlexNet', num_classes)
    # net = get_pretrained_model('GoogLeNet', num_classes)
    # net = get_pretrained_model('RegNet_Y_32GF', num_classes)

    #Device setup
    device = torch.device("cpu")  # Default device is CPU
    if torch.cuda.is_available():
        device = torch.device("cuda")  # Switch to GPU if available

    net = net.to(device)  # Move the model to the selected device

    #Stuff setup
    # Criterion (loss function)
    criterion = nn.CrossEntropyLoss(weight=class_weights)

    # criterion = nn.CrossEntropyLoss()
    criterion = criterion.to(device)

    # Adam optimizer
    optimizer = optim.Adam(net.parameters(), lr=1e-5)



    #Training the network
    # number of loops over the dataset
    num_epochs = 30
    accuracy = []
```

```
    val_accuracy = []
    losses = []
    val_losses = []


    def get_accuracy(predicted, labels):
        batch_len = labels.size(0)
        correct = (predicted == labels).sum().item()
        return batch_len, correct

    def evaluate(model, val_loader):
        losses = 0
        num_samples_total = 0
        correct_total = 0
        model.eval()
        for inputs, labels in val_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            out = model(inputs)
            _, predicted = torch.max(out, 1)
            loss = criterion(out, labels)
            losses += loss.item()
            b_len, corr = get_accuracy(predicted, labels)
            num_samples_total += b_len
            correct_total += corr
        accuracy = correct_total / num_samples_total
        losses = losses / len(val_loader)
        return losses, accuracy

    for epoch in range(num_epochs):
        running_loss = 0.0
        correct_total = 0.0
        num_samples_total = 0.0
        for i, data in enumerate(train_data_loader):
            # get the inputs
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(
                device)  # Move inputs and labels to the same device as the
model
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
```

```
            loss.backward()
            optimizer.step()

            # compute accuracy
            _, predicted = torch.max(outputs, 1)
            b_len, corr = get_accuracy(predicted, labels)
            num_samples_total += b_len
            correct_total += corr
            running_loss += loss.item()

        running_loss /= len(train_data_loader)
        train_accuracy = correct_total / num_samples_total
        val_loss, val_acc = evaluate(net, validation_data_loader)

        print('Epoch: %d' % (epoch + 1))
        print('Loss: %.3f  Accuracy:%.3f' % (running_loss, train_accuracy))
        print('Validation  Loss:  %.3f   Val  Accuracy:  %.3f'  %  (val_loss,
val_acc))

        losses.append(running_loss)
        val_losses.append(val_loss)
        accuracy.append(train_accuracy)
        val_accuracy.append(val_acc)
    print('Finished Training')

    epoch = range(1, num_epochs + 1)

    # Plot the Loss curves
    fig = plt.figure(figsize=(10, 15))
    plt.subplot(2, 1, 2)
    plt.plot(epoch, losses, label='Training loss')
    plt.plot(epoch, val_losses, label='Validation loss')
    plt.title('Training and Validation Loss')
    plt.xlabel('Epochs')
    plt.legend()
    plt.figure()
    plt.show()

    # Plot the Accuracy curves
    fig = plt.figure(figsize=(10, 15))
    plt.subplot(2, 1, 2)
    plt.plot(epoch, accuracy, label='Training accuracy')
    plt.plot(epoch, val_accuracy, label='Validation accuracy')
```

```
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.legend()
plt.figure()
plt.show()

#8-------------------------------
#Evaluating the network

#fig = plt.figure(figsize=(10, 15))
dataiter = iter(test_data_loader)
images, labels = next(dataiter)

imshow(torchvision.utils.make_grid(images))

correct = 0
total = 0
net.eval()
with torch.no_grad():
    for data in test_data_loader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

# Overall accuracy
print('\nAccuracy of the network on the test images: %d %%' % (
        100 * correct / total))

print("\nTest data")
class_correct = list(0. for i in range(len(classes)))
class_total = list(1e-7 for i in range(len(classes)))
with torch.no_grad():
    for data in test_data_loader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(3):
            label = labels[i]
```

```
            class_correct[label] += c[i].item()
            class_total[label] += 1

for i in range(len(classes)):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))

print('\nValidation data')

class_correct = list(0. for i in range(len(classes)))
class_total = list(1e-7 for i in range(len(classes)))
net.eval()
with torch.no_grad():
    for data in validation_data_loader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(3):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1

for i in range(len(classes)):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))

# print('custom data')
# # Accuracy for each class
# for i in range(num_classes):
#     class_accuracy = 100 * class_correct[i] / class_total[i]
#     print('Accuracy of %s : %d %%' % (classes[i], class_accuracy))

confusion_matrix = torch.zeros(len(classes), len(classes))
with torch.no_grad():
    for data in test_data_loader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        for t, p in zip(labels.view(-1), predicted.view(-1)):
            confusion_matrix[t.long(), p.long()] += 1
```

```
    cm = confusion_matrix.numpy()

    fig, ax = plt.subplots(figsize=(7, 7))
    sns.heatmap(cm / (cm.astype(float).sum(axis=1) + 1e-9), annot=False,
ax=ax)

    # labels, title and ticks
    ax.set_xlabel('Predicted', size=25);
    ax.set_ylabel('True', size=25);
    ax.set_title('Confusion Matrix(Test data)', size=25);
    ax.xaxis.set_ticklabels(['akiec', 'bcc', 'bkl', 'df', 'mel', 'nv',
'vasc'], size=15); \
            ax.yaxis.set_ticklabels(['akiec', 'bcc', 'bkl', 'df', 'mel',
'nv', 'vasc'], size=15);

    plt.show()

    confusion_matrix = torch.zeros(len(classes), len(classes))
    with torch.no_grad():
        for data in validation_data_loader:
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            outputs = net(images)
            _, predicted = torch.max(outputs, 1)
            for t, p in zip(labels.view(-1), predicted.view(-1)):
                confusion_matrix[t.long(), p.long()] += 1

    cm = confusion_matrix.numpy()

    fig, ax = plt.subplots(figsize=(7, 7))
    sns.heatmap(cm / (cm.astype(float).sum(axis=1) + 1e-9), annot=False,
ax=ax)
    # labels, title and ticks
    ax.set_xlabel('Predicted', size=25);
    ax.set_ylabel('True', size=25);
    ax.set_title('Confusion Matrix(Validation data)', size=25);
    ax.xaxis.set_ticklabels(['akiec', 'bcc', 'bkl', 'df', 'mel', 'nv',
'vasc'], size=15); \
            ax.yaxis.set_ticklabels(['akiec', 'bcc', 'bkl', 'df', 'mel',
'nv', 'vasc'], size=15);

    plt.show()
```

ДОДАТОК В

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	100 с.
	Інші документи	
2.	Презентаційні матеріали	__ плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Малихін О.С.			Дослідження методів штучного інтелекту для аналізу дерматоскопічних зображень при діагностуванні захворювань шкіри	Шифр групи	Код напр./спец.
Перевірив		Селіванова К.Г.				СШІм-22-1	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Філатов В.О.					