ДОДАТОК А
ГРАФІЧНИЙ МАТЕРІАЛ КВАЛІФІКАЦІЙНОЇ РОБОТИ

2025

Харківський національний університет радіоелектроніки
Кафедра ЕОМ

Кваліфікаційна робота
Другий рівень(магістр)

Методикласифікації зображеньз використанням машинногонавчання

Автор
Прокопчик М.В.
ст. гр. СПм-23-5

Керівник
Ткачов В.М.
доц. каф. ЕОМ

20
25

Зміст

01
Огляд Предметної Області

Огляд Предметної області

- **Традиційні методи класифікації зображень**
  - Ручне виділення ознак (HOG, LBP, SIFT)
  - Алгоритми SVM, k-NN, Random Forest
  - Обмеження щодо гнучкості та адаптивності
- **Глибоке навчання: CNN**
  - Автоматичне виділення ознак
  - Популярні архітектури (AlexNet, VGG, ResNet)
  - Складнощі при класифікації малих зображень
  - Потреба в попередній обробці (super -resolution)
- **Моделі на основі трансформерів (Vision Transformers)**
  - Принцип глобальної обробки зображення
  - Високі вимоги до якості та обсягів даних
  - Нестабільність на низько роздільних зображеннях
- **Гібридні та комбіновані підходи**
  - Поєднання CNN + SVM або інших класичних моделей

02
Аналіз Існуючих Рішень

Аналіз існуючих рішень

CNN

Недоліки: втрата якості при зменшенні розміру об’єкту

ViT

Недоліки: необхідні великі набори даних для навчання

KNN та інші

Недоліки: необхідність дуже гарних вхідних даних

Гібрид CNN + ViT

Складка архітектура та велике використання CPU

Актуальність Обраної Теми

Використання класифікації зображень

Класифікації зображень використовується у:
медеціні, сфера безпеки, виробництві та у військовому ділі

Класифікації БПЛА

БПЛА інтегруються в різні сфери життєдіяльності людини від БПЛА-доставників та БПЛА для зйомки до БПЛА-розвідників та БПЛА-бомбардувальників

Різні інциденти

Невідомі БПЛА почали заважати аеропортам ще з 2015 року, коли вони влітали в їх пространство, а зараз вони несуть ще більшу загрозу.

Методи Класифікації Зображень з використанням МН

03
Мета та Задачі Дослідження

Мета Кваліфікаційної роботи

Розробка методу класифікації БПЛА з різною якістю як зображення, так і розміром самого об’єекту класифікації базуючих на базових методах класифікації. Оптимізації використовуючих ресурсів під час класифікації

Методи Класифікації Зображень з використанням МН

Задачі Кваліфікаційної роботи

- Аналіз оптимального базового методу класифікації
- Виявлення проблеми базового методу та покращення його. Реалізація оновленого методу
- Порівняння з базовими методами в різних умовах
- Пошук та аналіз оптимізації роботи метода та реалізації
- Проведення емпіричних дослідів
- Аналіз отриманих результатів дослідження

03.1

Аналіз базових методів класифікації; виявлення проблем в архітектурі

Порівняння базових методів

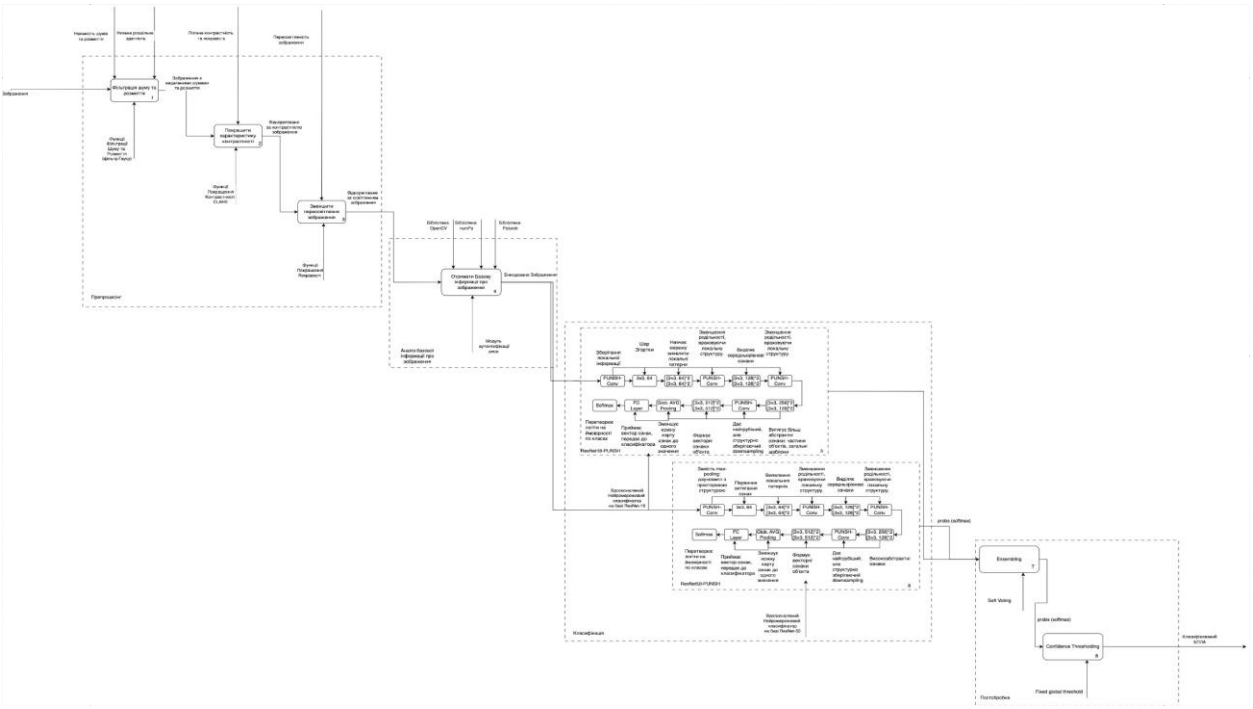
Критерій	CNN	RNN	ViT	KNN
Тип даних	Просторові	Послідовні	Просторові, подані як послідовність патчів	Просторові
Ієрархія ознак	Присутня	Відсутня	Відсутня	Відсутня
Узагальнення на малих наборах даних	Високе	Низьке	Низьке	Високе
Обчислювальна ефективність	Висока	Низька	Низька	Низька

Виявлення проблеми в CNN архітектурі

Назва Моделі	Відсоток помилки	Назва набіру
LeNet-5	9.15%	MNIST
AlexNet-8	12.34%	ImageNet
GoogleNet-22	8.71%	ImageNet, Place365
VGGNet-19	7.70%	ImageNet
ResNet-50	7.29%	ImageNet

Всі моделі використовували наступні дані для навчання:

- *використання оптимізатора(SGD, початкова швидкість навчання 0.1)*
- *аугментація зображень*
- *регулізація вісів*



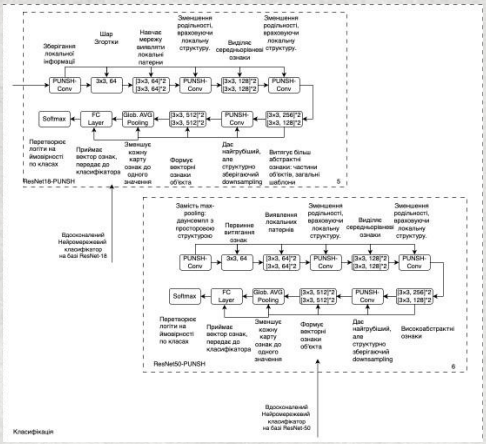
Дослідження оптимальної архітектури на базі CNN

- Додавання алгоритмів постобробки вхідних зображень
- Зміна архітектури ResNet для поліпшення класифікації неякісних зображень
- Модуль оптимізації використання CPU ресурсів
- Модуль постобробки результатів класифікації
- Реалізації класифікаторів на основі зміненої архітектури ResNet

Методи Класифікації Зображень з використанням МН

03.2
Порівняння з базовими методами класифікації

Класифікатор на основі архітектури ResNet



Використані Методи Аугментації

Результати порівняння класифікації

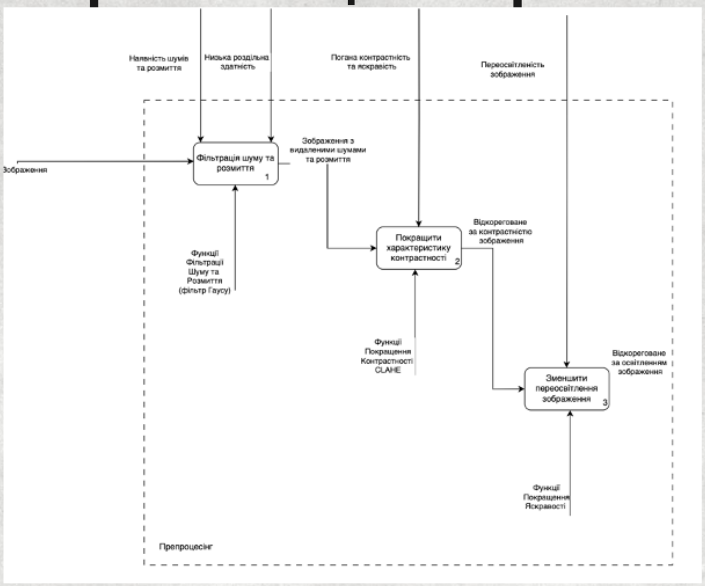
Метод	Набір Даних	Точність
ResNet18-PUNSH	DICD	50.13%
ResNet18	DICD	43.68%
ResNet50-PUNSH	DICD	87.01%
Convolutional Nystromformer for Vision	DICD	48.03%
RNN	DICD	47.17%
KNN	DICD	79.38%
ViT	DICD	67.70%

Методи Класифікації Зображень з використанням МН

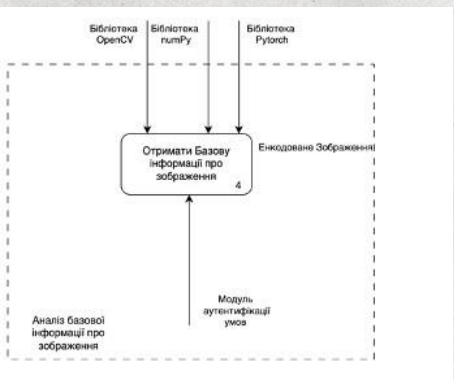
03.3
Пошук Оптимального класифікатора

Оптимізації використання
ресурсів CPU для класифікації

Нормалізації зображення



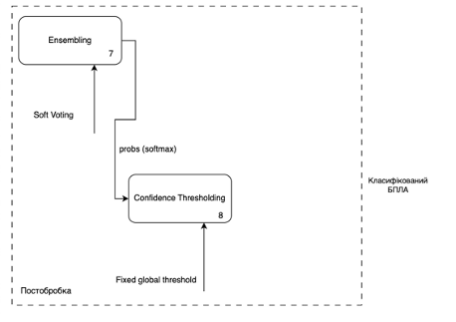
Оптимізації використання CPU



Алгоритм аналізу базової інформації описан далі:

- приведення зображення до уніфікованого формату;
- оцінка різкості зображення;
- оцінка рівня шуму;
- оцінка експозиції;
- оцінка контрастності;
- формування вектору ознак якості зображення;
- класифікація рівня якості або типу зображення;
- вибір відповідної моделі класифікації.

Пост обробка результатів зображення



Методи Класифікації Зображень з використанням МН

03.4
Порівняння оптимізованого методу з базовими

Результати Порівняння

Метод/Модель	Збільшення використання CPU, у відстоках, середне значення	Точність
ResNet50-PUNSH	22.4%	95.37%
ResNet18-PUNSH	9.37%	65.50%
Запропонований метод	15.32%	95.30%
ViT	29.21%	67.70%
KNN	9.37%	79.38%
RNN	27.61%	47.17%

Методи Класифікації Зображень з використанням МН

04

Висновки

Висновки

В результаті написання кваліфікаційної роботи досліджено та розроблено метод класифікації зображень на основі CNN/ResNet архітектури для забезпечення кращої якості класифікації зображень з малими об’єктами для класифікації, наприклад БПЛА, чи поганої якості. Оптимізовано використання CPU під час класифікації без втрати якості.

Вирішені наступні задачі

- Проведено аналіз та дослідження класичних методів та моделей на базі CNN
- Виявлено недолік CNN архітектури для класифікації неякісних зображень
- Створено класифікатор на основі ResNet архітектури
- Розроблено модуль оптимізації використання CPU та нормалізації вхідних зображень
- Створено модуль постобробки результатів класифікації
- Проведено емпіричне дослідження якості класифікації у порівняння з базовими методами
- Проаналізовано отримані результати

Методи Класифікації Зображень з використанням МН

05

Подальший розвиток проекту

Подальші дослідження та покращення

- Поточний метод використовую поліпшену, але базову архітектуру CNN/ResNet; можливий перехід на гібридну архітектуру для класифікатора на основі CNN + ViT методів з реалізацією генерування даних для навчання
- Можливе створення IoT/Embedded рішення для використання запропонованого методу
- Розгортання методу у Хмарі/Оркестраторі для підвищення доступності моделі

Методи Класифікації Зображень з використанням МН

06

Апробація

ДОДАТОК Б

ЛІСТИНГ КОДУ ФІЛЬТРАЦІЇ ЗОБРАЖЕННЯ ВИКОРИСТОВУЮЧИ ФУНКЦІЮ ГАУСУ

```
import cv2
import numpy as np
import os

def filter_blur_and_noises(input_path: str, output_path: str,
                  kernel_size: int, boost_blue: bool,
blue_intensity: float):
    """
    Processes the image with blur, edge enhancement, and
optional blue boost.
    """
    image = cv2.imread(input_path)
    if image is None:
        raise ValueError(f"Could not load image: {input_path}")

    if len(image.shape) == 2 or image.shape[2] == 1:
        image = cv2.cvtColor(image, cv2.COLOR_GRAY2BGR)


    blurred = cv2.GaussianBlur(image, (kernel_size,
kernel_size), 0)


    sharpen_kernel = np.array([[0, -1, 0],
                               [-1, 5, -1],
                               [0, -1, 0]])
    sharpened = cv2.filter2D(blurred, -1, sharpen_kernel)


    if boost_blue:
        b, g, r = cv2.split(sharpened)
        b = cv2.addWeighted(b, blue_intensity, b, 0, 0)
        result = cv2.merge([b, g, r])
    else:
        result = sharpened

    os.makedirs(os.path.dirname(output_path), exist_ok=True)
    cv2.imwrite(output_path, result)
    print(f"Processed image saved to: {output_path}")

def prefilter_blur_and_noises(input_path: str, output_path:
str):
    """
    Pre-analyzes the image and calls filter_blur_and_noises with
```

```
inferred parameters.
    """
    image = cv2.imread(input_path)
    if image is None:
        raise ValueError(f"Could not load image: {input_path}")

    if len(image.shape) == 2 or image.shape[2] == 1:
        image = cv2.cvtColor(image, cv2.COLOR_GRAY2BGR)


    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    lap_var = cv2.Laplacian(gray, cv2.CV_64F).var()


    if lap_var < 50:
        kernel_size = 7
    elif lap_var < 150:
        kernel_size = 5
    else:
        kernel_size = 3
    if kernel_size % 2 == 0:
        kernel_size += 1


    b, g, r = cv2.split(image)
    blue_mean = np.mean(b)
    rg_mean = (np.mean(r) + np.mean(g)) / 2

    boost_blue = blue_mean < rg_mean * 0.95
    blue_intensity = min(1.0 + (rg_mean - blue_mean) / 100, 1.5)
if boost_blue else 1.0

    print(f"[AUTO] Noise Level: {lap_var:.2f}, Kernel:
{kernel_size}, Boost Blue: {boost_blue}, Blue Intensity:
{blue_intensity:.2f}")


    filter_blur_and_noises(input_path, output_path, kernel_size,
boost_blue, blue_intensity)
```

ДОДАТОК В

МОДЕЛЬ НА ОСНОВІ RESNET50-PUNSH

```
import torch
import torch.nn as nn


class space_to_depth(nn.Module):
    # Changing the dimension of the Tensor
    def __init__(self, dimension=1):
        super().__init__()
        self.d = dimension

    def forward(self, x):
        return torch.cat([x[..., ::2, ::2], x[..., 1::2, ::2],
x[..., ::2, 1::2], x[..., 1::2, 1::2]], 1)

def autopad(k, p=None):  # kernel, padding
    # Pad to 'same'
    if p is None:
        p = k // 2 if isinstance(k, int) else [x // 2 for x in
k]  # auto-pad
    return p


class Conv(nn.Module):
    # Standard convolution
    def __init__(self, c1, c2, k=1, s=1, p=None, g=1, act=True):
# ch_in, ch_out, kernel, stride, padding, groups
        super().__init__()
        self.conv = nn.Conv2d(c1, c2, k, s, autopad(k, p),
groups=g, bias=False)
        self.bn = nn.BatchNorm2d(c2)
        self.act = nn.SiLU() if act is True else (act if
isinstance(act, nn.Module) else nn.Identity())

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

    def forward_fuse(self, x):
        return self.act(self.conv(x))


class Focus(nn.Module):
    # Focus wh information into c-space
    def __init__(self, c1, c2, k=1, s=1, p=None, g=1, act=True):
# ch_in, ch_out, kernel, stride, padding, groups
        super().__init__()
        self.conv = Conv(c1 * 4, c2, k, s, p, g, act)
        # self.contract = Contract(gain=2)
```

```
    def forward(self, x):  # x(b,c,w,h) -> y(b,4c,w/2,h/2)
        return self.conv(torch.cat([x[..., ::2, ::2], x[...,
1::2, ::2], x[..., ::2, 1::2], x[..., 1::2, 1::2]], 1))
        # return self.conv(self.contract(x))


class BottleNeck(nn.Module):
    """Residual block for resnet over 50 layers

    """
    expansion = 4
    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()
        layers = [
                  nn.Conv2d(in_channels, out_channels,
kernel_size=1, bias = False),
                  nn.BatchNorm2d(out_channels),
                  nn.ReLU(inplace=True),
        ]

        if stride ==2:

            layers2 = [
            nn.Conv2d(out_channels, out_channels,stride= 1,
kernel_size=3, padding=1, bias= False),
            space_to_depth(),   # the output of this will result
in 4*out_channels
            nn.BatchNorm2d(4*out_channels),
            nn.ReLU(inplace=True),

            nn.Conv2d(4*out_channels, out_channels*
BottleNeck.expansion, kernel_size=1, bias = False),
            nn.BatchNorm2d(out_channels * BottleNeck.expansion),
            ]

        else:

            layers2 = [
            nn.Conv2d(out_channels, out_channels,stride= stride,
kernel_size=3, padding=1, bias= False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),

            nn.Conv2d(out_channels, out_channels*
BottleNeck.expansion, kernel_size=1, bias = False),
            nn.BatchNorm2d(out_channels * BottleNeck.expansion),
            ]

        layers.extend(layers2)

        self.residual_function = torch.nn.Sequential(*layers)
```

```
        self.shortcut = nn.Sequential()

        if stride != 1 or in_channels != out_channels *
BottleNeck.expansion:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels *
BottleNeck.expansion, stride=stride, kernel_size=1, bias=False),
                nn.BatchNorm2d(out_channels *
BottleNeck.expansion)
            )

    def forward(self, x):
        return nn.ReLU(inplace=True)(self.residual_function(x) +
self.shortcut(x))

class ResNet(nn.Module):

    def __init__(self, block, num_block, num_classes=100):
        super().__init__()

        self.in_channels = 64

        self.conv1 = Focus(3, 64, k=1,s=1)

        self.conv2_x = self._make_layer(block, 64, num_block[0],
1)
        self.conv3_x = self._make_layer(block, 128,
num_block[1], 2)
        self.conv4_x = self._make_layer(block, 256,
num_block[2], 2)
        self.conv5_x = self._make_layer(block, 512,
num_block[3], 2)
        self.avg_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * block.expansion, num_classes)

    def _make_layer(self, block, out_channels, num_blocks,
stride):

        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_channels, out_channels,
stride))
            self.in_channels = out_channels * block.expansion

        return nn.Sequential(*layers)

    def forward(self, x):
        output = self.conv1(x)
        output = self.conv2_x(output)
        output = self.conv3_x(output)
        output = self.conv4_x(output)
```

```
        output = self.conv5_x(output)
        output = self.avg_pool(output)
        output = output.view(output.size(0), -1)
        output = self.fc(output)

        return output


def resnet50():
    """ return a ResNet 50 object
    """
    return ResNet(BottleNeck, [3, 4, 6, 3])



if __name__ =="__main__":
     net = resnet50()
     x = torch.empty((2,3,112,112)).normal_()
     print(net(x).shape)
```

ДОДАТОК Г

Код методу IAGCWD

```
import cv2
import glob
import argparse
import numpy as np
from matplotlib import pyplot as plt
from scipy.linalg import fractional_matrix_power

def image_agcwd(img, a=0.25, truncated_cdf=False):
    h,w = img.shape[:2]
    hist,bins = np.histogram(img.flatten(),256,[0,256])
    cdf = hist.cumsum()
    cdf_normalized = cdf / cdf.max()
    prob_normalized = hist / hist.sum()

    unique_intensity = np.unique(img)
    intensity_max = unique_intensity.max()
    intensity_min = unique_intensity.min()
    prob_min = prob_normalized.min()
    prob_max = prob_normalized.max()

    pn_temp = (prob_normalized - prob_min) / (prob_max -
prob_min)
    pn_temp[pn_temp>0] = prob_max * (pn_temp[pn_temp>0]**a)
    pn_temp[pn_temp<0] = prob_max * (-((-
pn_temp[pn_temp<0])**a))
    prob_normalized_wd = pn_temp / pn_temp.sum() # normalize to
[0,1]
    cdf_prob_normalized_wd = prob_normalized_wd.cumsum()

    if truncated_cdf:
        inverse_cdf = np.maximum(0.5,1 - cdf_prob_normalized_wd)
    else:
        inverse_cdf = 1 - cdf_prob_normalized_wd

    img_new = img.copy()
    for i in unique_intensity:
        img_new[img==i] = np.round(255 * (i /
255)**inverse_cdf[i])

    return img_new

def process_bright(img):
    img_negative = 255 - img
    agcwd = image_agcwd(img_negative, a=0.25,
truncated_cdf=False)
    reversed = 255 - agcwd
    return reversed
```

```
def process_dimmed(img):
    agcwd = image_agcwd(img, a=0.75, truncated_cdf=True)
    return agcwd

def main():
    parser = argparse.ArgumentParser(description='IAGCWD')
    parser.add_argument('--input', dest='input_dir',
default='./input/', type=str, \
                        help='Input directory for image(s)')
    parser.add_argument('--output', dest='output_dir',
default='./output/', type=str, \
                        help='Output directory for image(s)')
    args = parser.parse_args()

    img_paths = glob.glob(args.input_dir+'*')
    for path in img_paths:
        img = cv2.imread(path, 1)
        name = path.split('\\')[-1].split('.')[0]

        # Extract intensity component of the image
        YCrCb = cv2.cvtColor(img, cv2.COLOR_BGR2YCrCb)
        Y = YCrCb[:,:,0]
        # Determine whether image is bright or dimmed
        threshold = 0.3
        exp_in = 112 # Expected global average intensity
        M,N = img.shape[:2]
        mean_in = np.sum(Y/(M*N))
        t = (mean_in - exp_in)/ exp_in

        # Process image for gamma correction
        img_output = None
        if t < -threshold: # Dimmed Image
            print (name + ": Dimmed")
            result = process_dimmed(Y)
            YCrCb[:,:,0] = result
            img_output = cv2.cvtColor(YCrCb,cv2.COLOR_YCrCb2BGR)
        elif t > threshold:
            print (name + ": Bright Image") # Bright Image
            result = process_bright(Y)
            YCrCb[:,:,0] = result
            img_output = cv2.cvtColor(YCrCb,cv2.COLOR_YCrCb2BGR)
        else:
            img_output = img

        cv2.imwrite(args.output_dir+name+'.jpg', img_output)
```

ДОДАТОК Ґ

Код для навчання НМ

```
from ptflops import get_model_complexity_info
import os
import sys
import argparse
import time
from datetime import datetime

import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms

import wandb
wandb.init()

from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter

from conf import settings
from   utils   import   get_network,   get_training_dataloader,
get_test_dataloader, WarmUpLR, \
    most_recent_folder,     most_recent_weights,     last_epoch,
best_acc_weights

def train(epoch):

    start = time.time()
    net.train()
    for       batch_index,       (images,       labels)       in
enumerate(cifar10_training_loader):

        if args.gpu:
            labels = labels.cuda()
            images = images.cuda()

        optimizer.zero_grad()
        outputs = net(images)
        loss = loss_function(outputs, labels)
        loss.backward()
        optimizer.step()

        n_iter  =  (epoch  -  1)  *  len(cifar10_training_loader)  +
batch_index + 1
```

```
        last_layer = list(net.children())[-1]
        for name, para in last_layer.named_parameters():
            if 'weight' in name:

writer.add_scalar('LastLayerGradients/grad_norm2_weights',
para.grad.norm(), n_iter)
            if 'bias' in name:

writer.add_scalar('LastLayerGradients/grad_norm2_bias',
para.grad.norm(), n_iter)

        print('Training              Epoch:              {epoch}
[{trained_samples}/{total_samples}]\tLoss:           {:0.4f}\tLR:
{:0.6f}'.format(
            loss.item(),
            optimizer.param_groups[0]['lr'],
            epoch=epoch,
            trained_samples=batch_index * args.b + len(images),
            total_samples=len(cifar10_training_loader.dataset)
        ))

        wandb.log({'loss': loss.item()})

        #update training loss for each iteration
        writer.add_scalar('Train/loss', loss.item(), n_iter)

        if epoch <= args.warm:
            warmup_scheduler.step()

    for name, param in net.named_parameters():
        layer, attr = os.path.splitext(name)
        attr = attr[1:]
        writer.add_histogram("{}/{}".format(layer,  attr),  param,
epoch)

    finish = time.time()

    print('epoch      {}      training      time      consumed:
{:.2f}s'.format(epoch, finish - start))

@torch.no_grad()
def eval_training(epoch=0, tb=True):

    start = time.time()
    net.eval()

    test_loss = 0.0 # cost function error
    correct = 0.0

    for (images, labels) in cifar10_test_loader:

        if args.gpu:
```

```
            images = images.cuda()
            labels = labels.cuda()

        outputs = net(images)
        loss = loss_function(outputs, labels)

        test_loss += loss.item()
        _, preds = outputs.max(1)
        correct += preds.eq(labels).sum()

    finish = time.time()
    if args.gpu:
        print('GPU INFO.....')
        print(torch.cuda.memory_summary(), end='')
    print('Evaluating Network.....')
    print('Test set: Epoch: {}, Average loss: {:.4f}, Accuracy:
{:.4f}, Time consumed:{:.2f}s'.format(
        epoch,
        test_loss / len(cifar10_test_loader.dataset),
        correct.float() / len(cifar10_test_loader.dataset),
        finish - start
    ))

    wandb.log({'Test          loss':          test_loss          /
len(cifar10_test_loader.dataset)})
    wandb.log({'Test         Accuracy':correct.float()          /
len(cifar10_test_loader.dataset)})

    print()

    #add informations to tensorboard
    if tb:
        writer.add_scalar('Test/Average   loss',   test_loss   /
len(cifar10_test_loader.dataset), epoch)
        writer.add_scalar('Test/Accuracy',   correct.float()   /
len(cifar10_test_loader.dataset), epoch)

    return correct.float() / len(cifar10_test_loader.dataset)

if __name__ == '__main__':

    parser = argparse.ArgumentParser()
    parser.add_argument('-net',     type=str,     required=True,
help='net type')
    parser.add_argument('-gpu',              action='store_true',
default=False, help='use gpu or not')
    parser.add_argument('-b', type=int, default=128, help='batch
size for dataloader')
    parser.add_argument('-warm', type=int, default=1, help='warm
up training phase')
    parser.add_argument('-lr',      type=float,      default=0.1,
help='initial learning rate')
```

```
    parser.add_argument('-resume',          action='store_true',
default=False, help='resume training')
    args = parser.parse_args()

    net = get_network(args)

    macs, params = get_model_complexity_info(net, (3, 32, 32),
as_strings=True,

print_per_layer_stat=True, verbose=True)
    print('{:<30}   {:<8}'.format('Computational complexity: ',
macs))
    print('{:<30}    {:<8}'.format('Number of parameters:  ',
params))


    #data preprocessing:
    cifar10_training_loader = get_training_dataloader(
        settings.CIFAR100_TRAIN_MEAN,
        settings.CIFAR100_TRAIN_STD,
        num_workers=4,
        batch_size=args.b,
        shuffle=True
    )

    cifar10_test_loader = get_test_dataloader(
        settings.CIFAR100_TRAIN_MEAN,
        settings.CIFAR100_TRAIN_STD,
        num_workers=4,
        batch_size=args.b,
        shuffle=True
    )

    loss_function = nn.CrossEntropyLoss()
    optimizer    =    optim.SGD(net.parameters(),    lr=args.lr,
momentum=0.9, weight_decay=5e-4)
    train_scheduler  =  optim.lr_scheduler.MultiStepLR(optimizer,
milestones=settings.MILESTONES, gamma=0.2) #learning rate decay
    iter_per_epoch = len(cifar10_training_loader)
    warmup_scheduler  =  WarmUpLR(optimizer,  iter_per_epoch  *
args.warm)

    if args.resume:
        recent_folder                                          =
most_recent_folder(os.path.join(settings.CHECKPOINT_PATH,
args.net), fmt=settings.DATE_FORMAT)
        if not recent_folder:
            raise Exception('no recent folder were found')

        checkpoint_path = os.path.join(settings.CHECKPOINT_PATH,
args.net, recent_folder)

    else:
```

```
        checkpoint_path = os.path.join(settings.CHECKPOINT_PATH,
args.net, settings.TIME_NOW)

    #use tensorboard
    if not os.path.exists(settings.LOG_DIR):
        os.mkdir(settings.LOG_DIR)

    #since tensorboard can't overwrite old values
    #so the only way is to create a new tensorboard log
    writer = SummaryWriter(log_dir=os.path.join(
            settings.LOG_DIR, args.net, settings.TIME_NOW))
    input_tensor = torch.Tensor(1, 3, 32, 32)
    if args.gpu:
        input_tensor = input_tensor.cuda()
    writer.add_graph(net, input_tensor)

    #create checkpoint folder to save model
    if not os.path.exists(checkpoint_path):
        os.makedirs(checkpoint_path)
    checkpoint_path = os.path.join(checkpoint_path, '{net}-
{epoch}-{type}.pth')

    best_acc = 0.0
    if args.resume:
        best_weights                                          =
best_acc_weights(os.path.join(settings.CHECKPOINT_PATH,
args.net, recent_folder))
        if best_weights:
            weights_path                                      =
os.path.join(settings.CHECKPOINT_PATH, args.net, recent_folder,
best_weights)
            print('found best acc weights
file:{}'.format(weights_path))
            print('load best training file to test acc...')
            net.load_state_dict(torch.load(weights_path))
            best_acc = eval_training(tb=False)
            print('best acc is {:0.2f}'.format(best_acc))

        recent_weights_file                                   =
most_recent_weights(os.path.join(settings.CHECKPOINT_PATH,
args.net, recent_folder))
        if not recent_weights_file:
            raise Exception('no recent weights file were found')
        weights_path = os.path.join(settings.CHECKPOINT_PATH,
args.net, recent_folder, recent_weights_file)
        print('loading weights file {} to resume
training.....'.format(weights_path))
        net.load_state_dict(torch.load(weights_path))

        resume_epoch                                          =
last_epoch(os.path.join(settings.CHECKPOINT_PATH, args.net,
recent_folder))
```

```
    for epoch in range(1, settings.EPOCH + 1):
        if epoch > args.warm:
            train_scheduler.step(epoch)

        if args.resume:
            if epoch <= resume_epoch:
                continue

        train(epoch)
        acc = eval_training(epoch)

        #start to save best performance model after learning
rate decay to 0.01
        if epoch > settings.MILESTONES[1] and best_acc < acc:
            weights_path = checkpoint_path.format(net=args.net,
epoch=epoch, type='best')
            print('saving weights file to
{}'.format(weights_path))
            torch.save(net.state_dict(), weights_path)
            best_acc = acc
            continue

        if not epoch % settings.SAVE_EPOCH:
            weights_path = checkpoint_path.format(net=args.net,
epoch=epoch, type='regular')
            print('saving weights file to
{}'.format(weights_path))
            torch.save(net.state_dict(), weights_path)

    writer.close()
```

ДОДАТОК Д

Отримання інформації про використання ресурсів під час роботи RESNET50-PUNSH

```
import sys
import argparse
import torch
from PIL import Image
from torchvision import transforms
import psutil
import time
import types

from utils import get_network


def make_args():
    args = types.SimpleNamespace()
    args.net = 'resnet50_spd'
    args.gpu = False  # Set to True if using GPU
    return args

def load_model(model_path):
    args = make_args()
    model = get_network(args)
    state_dict = torch.load(model_path, map_location='cpu')
    model.load_state_dict(state_dict)
    model.eval()
    return model

def preprocess_image(image_path):
    transform = transforms.Compose([
        transforms.Resize((32, 32)),  #
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465),
                             (0.2023, 0.1994, 0.2010))
    ])
    image = Image.open(image_path).convert('RGB')
    return transform(image).unsqueeze(0)

def

def classify(name, method, input_tensor):
    cpu_before = psutil.cpu_percent()
    start = time.time()
    with torch.no_grad():
        output = model(input_tensor)
    end = time.time()
    cpu_after = psutil.cpu_percent()
    predicted_class = output.argmax(dim=1).item()
```

```
    print(f"Using {name} method")
    print(f"Predicted class index: {predicted_class}")
    print(f"CPU usage before/after: {cpu_before}% →
{cpu_after}%")
    print(f"Inference time: {end - start:.4f} seconds")

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("images_path")
    parser.add_argument("model_path_50")
    parser.add_argument("model_path_18")
    args = parser.parse_args()

    model_18 = load_model(args.model_path_18)
    model_50 = load_model(args.model_path_50)

    input_tensor = preprocess_images(args.images_path)
    method_pipeline = Method(model_path=model_path_50,
pipeline=True)
    classify(method_pipeline, input_tensor)

    method_18 = Method(model_path=model_path_18)
    classify(method_18, input_tensor)

    method_50 = Method(model_path=model_path_50)
    classify(method_50, input_tensor)


if __name__ == "__main__":
    main()
```

ДОДАТОК Е

Модуль отримання та класифікації якості зображення на основі базової інформації про нього

```
import cv2
import numpy as np
from sklearn.preprocessing import StandardScaler
import joblib
import torch
import torch.nn.functional as F

def preprocess_image(img_path, size=(224, 224)):
    img = cv2.imread(img_path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img = cv2.resize(img, size)
    return img

def estimate_sharpness(img_gray):
    laplacian_var = cv2.Laplacian(img_gray, cv2.CV_64F).var()
    return laplacian_var

def estimate_noise(img_gray):
    noise = np.mean(np.abs(img_gray[:, :-1] - img_gray[:, 1:]))
    return noise

def estimate_exposure(img_gray):
    hist = cv2.calcHist([img_gray], [0], None, [256], [0,
256]).flatten()
    total_pixels = img_gray.size
    dark = np.sum(hist[:30]) / total_pixels
    bright = np.sum(hist[225:]) / total_pixels
    return dark, bright

def estimate_contrast(img_gray):
    contrast = img_gray.std()
    return contrast

def extract_features(img):
    img_gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    features = [
        estimate_sharpness(img_gray),
        estimate_noise(img_gray),
        *estimate_exposure(img_gray),
        estimate_contrast(img_gray)
    ]
    return np.array(features)
```

```
def classify_quality(features):
    sharpness, noise, dark, bright, contrast = features
    if sharpness < 20 or contrast < 10 or noise > 15:
        return "low"
    elif dark > 0.3 or bright > 0.3:
        return "medium"
    else:
        return "high"

def select_model(quality_label, model_paths):
    if quality_label == "high":
        model = joblib.load(model_paths["high"])
        return model
    elif quality_label == "low":
        model = joblib.load(model_paths["low"])
        return model
    else:  # medium
        model_high = joblib.load(model_paths["high"])
        model_low = joblib.load(model_paths["low"])
        return (model_high, model_low)

def run_pipeline(img_path, model_paths):
    img = preprocess_image(img_path)
    features = extract_features(img)
    quality = classify_quality(features)
    models = select_model(quality, model_paths)

    img_input = img.astype(np.float32) / 255.0
    img_input = np.transpose(img_input, (2, 0, 1))  # CHW
    img_input = np.expand_dims(img_input, axis=0)  # batch
    img_tensor = torch.tensor(img_input)

    if isinstance(models, tuple):
        model_high, model_low = models
        model_high.eval()
        model_low.eval()
        with torch.no_grad():
            out_high = F.softmax(model_high(img_tensor), dim=1)
            out_low = F.softmax(model_low(img_tensor), dim=1)
            combined_output = (out_high + out_low) / 2.0
            prediction = combined_output.argmax(dim=1).item()
    else:
        models.eval()
        with torch.no_grad():
            output = models(img_tensor)
            prediction = output.argmax(dim=1).item()

    return prediction, quality
```

ДОДАТОК Є

Сніпет коду базового методу VIT

```
import torch
from torch._C import dtype
import torch.nn as nn
from torch.nn.modules.conv import Conv2d
import copy
import argparse
import sys
import torch.nn.functional as F
from Utils import config

args = config.parse_args()

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

class VIT(nn.Module):
    def __init__(self, img_size=
(args.im_s,args.im_s),patch_size= (args.ps, args.ps), emb_dim =
args.emb_dim, mlp_dim= args.mlp_dim
,num_heads=args.num_heads,num_layers=args.num_layers,n_classes=2
, dropout_rate=0., at_d_r=args.at_d_r):
        super(VIT, self).__init__()

        self.nl = num_layers
        ih, iw = img_size
        ph, pw = patch_size
        num_patches = int((ih*iw)/(ph*pw))
        self.cls_tokens = nn.Parameter(torch.rand(1, 1,
emb_dim))
        self.patch_embed = Conv2d(in_channels=3,
                                  out_channels=emb_dim,
                                  kernel_size=patch_size,
                                  stride=patch_size)
        self.pos_embed = nn.Parameter(torch.randn(1, num_patches
+ 1, emb_dim))
        self.dropout = nn.Dropout(dropout_rate)

        self.tel = nn.ModuleList()
        for i in range(num_layers):
            layer = transencoder(emb_dim, mlp_dim, num_heads,
at_d_r)
            self.tel.append(layer)

        self.mlp_head = nn.Sequential(
            nn.LayerNorm(emb_dim),
```

```
            nn.Linear(emb_dim, n_classes)
        )
    def forward(self,x):
        x = self.patch_embed(x)
        x = x.permute(0, 2, 3, 1)
        b, h, w, c = x.shape
        x = x.reshape(b, h * w, c)
        cls_token = self.cls_tokens.repeat(b, 1, 1)
        x= torch.cat([cls_token, x], dim=1)
        embeddings = x + self.pos_embed
        embeddings = self.dropout(embeddings)
        for layer in self.tel:
            enc = layer(embeddings)
        mlp_head = self.mlp_head(enc[:, 0])
        return mlp_head

class transencoder(nn.Module):
    def __init__(self,emb_dim, mlp_dim, num_heads, at_d_r):
        super(transencoder, self).__init__()

        self.norm = nn.LayerNorm(emb_dim, eps=1e-6)
        self.mha = mha(emb_dim, num_heads, at_d_r)
        self.mlp = Mlp(emb_dim, mlp_dim)

    def forward(self, x):
        n = self.norm(x)
        attn = self.mha(n,n,n)
        output = attn+x
        n2 = self.norm(output)
        ff = self.mlp(n2)
        out = ff+output
        return out


class mha(nn.Module):
    def __init__(self, h_dim, n_heads, at_d_r):
        super().__init__()
        self.h_dim=h_dim
        self.linear = nn.Linear(h_dim, h_dim, bias=False)
        self.num_heads = n_heads
        self.norm = nn.LayerNorm(h_dim)
        self.dropout = nn.Dropout(at_d_r)
        self.softmax = nn.Softmax(dim=2)

    def forward(self, q, k ,v):
        rs = q.size()[0]
        batches, sequence_length, embeddings_dim = q.size()
        q1= nn.ReLU()(self.linear(q))
        k1= nn.ReLU()(self.linear(k))
        v1= nn.ReLU()(self.linear(v))

        q2 = torch.cat(torch.chunk(q1, self.num_heads, dim=2),
dim=0)
```

```
        k2 = torch.cat(torch.chunk(k1, self.num_heads, dim=2),
dim=0)
        v2 = torch.cat(torch.chunk(v1, self.num_heads, dim=2),
dim=0)

        outputs = torch.bmm(q2, k2.transpose(2, 1))
        outputs = outputs / (k2.size()[-1] ** 0.5)
        outputs = F.softmax(outputs, dim=-1)
        outputs = self.dropout(outputs)
        outputs = torch.bmm(outputs, v2)
        outputs = outputs.split(rs, dim=0)
        outputs = torch.cat(outputs, dim=2)
        outputs += outputs + q
        outputs = self.norm(outputs)
        return outputs

class Mlp(nn.Module):
    def __init__(self, emb_dim, mlp_dim, dropout_rate=0.):
        super(Mlp, self).__init__()
        self.fc1 = nn.Linear(emb_dim, mlp_dim)
        self.fc2 = nn.Linear(mlp_dim, emb_dim)
        self.act = nn.GELU()
        self.dropout= nn.Dropout(dropout_rate)


    def forward(self, x):

        out = self.fc1(x)
        out = self.act(out)
        out = self.dropout(out)
        out = self.fc2(out)
        out = self.dropout(out)
        return out
```

ДОДАТОК Ж

Сніпет коду базового методу KNN

```
import os
import cv2
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score


class KNNImageClassifier:
    def __init__(self, k=3, image_size=(32, 32)):
        self.k = k
        self.image_size = image_size
        self.model = KNeighborsClassifier(n_neighbors=k)

    def _load_images_from_folder(self, folder):
        data = []
        labels = []

        for label in os.listdir(folder):
            class_folder = os.path.join(folder, label)
            if not os.path.isdir(class_folder):
                continue
            for filename in os.listdir(class_folder):
                file_path = os.path.join(class_folder, filename)
                img = cv2.imread(file_path)
                if img is not None:
                    img = cv2.resize(img, self.image_size)
                    data.append(img.flatten())  # Flatten to 1D
vector
                    labels.append(label)

        return np.array(data), np.array(labels)

    def train(self, dataset_path):
        print("Loading data...")
        X, y = self._load_images_from_folder(dataset_path)
        print(f"Loaded {len(X)} images.")

        X_train, X_test, y_train, y_test = train_test_split(X,
y, test_size=0.2, random_state=42)
        print("Training KNN model...")
        self.model.fit(X_train, y_train)

        y_pred = self.model.predict(X_test)
        acc = accuracy_score(y_test, y_pred)
        print(f"Accuracy: {acc * 100:.2f}%")
```

```
    def predict(self, image_path):
        img = cv2.imread(image_path)
        if img is None:
            raise ValueError("Image not found or unreadable")
        img = cv2.resize(img,
self.image_size).flatten().reshape(1, -1)
        return self.model.predict(img)[0]
```