ICE2607 Lab 5: Pytorch & CNN

SJTU-SEIEE cny123222

December 26, 2024

1 Experiment Overview (实验概览)

Nowadays, neural networks play a crucial role in feature extraction and retrieval tasks in multimedia applications. In this experiment, we focus on utilizing neural networks for image feature extraction and retrieval, emphasizing their significance in modern multimedia retrieval systems.

The experiment is divided into two main parts. In the first part, a CIFAR-10 classifier was trained using a ResNet-20 architecture. We investigated the impact of different hyperparameters and image augmentation techniques in order to improve the classification accuracy.

In the second part, a pre-trained ResNet-50 model was employed for image retrieval on a custom dataset. Additionally, we explored the performance of the ViT-B/16 model in the same retrieval task.

2 Solution Approach (解决思路)

2.1 Part I: Image Classification

In the experiment, a ResNet-20 model was provided, requiring completion of a code snippet. Furthermore, enhancements were implemented in the codebase, aiming to facilitate a more convenient training process and improve the precision of image classification.

2.1.1 Completion of Test Accuracy Calculation

The calculation of test accuracy mirrors that of train accuracy, emphasizing the counting of accurately classified images and the total number of images. The provided code snippet illustrates this process.

```python
model.eval()
correct = 0 # number of correctly classified images
total = 0 # total number of images
with torch.no_grad():
    for batch_idx, (inputs, targets) in enumerate(testloader):
        outputs = model(inputs)
        _, predicted = outputs.max(1)
        total += targets.size(0)
        correct += predicted.eq(targets).sum().item()
test_accuracy = 100 * correct / total
```

```python
print(f"Epoch [{epoch}] - Test Accuracy: {test_accuracy:.3f}%")
```

2.1.2 Improvements to Training Process

- **Utilizing AdamW over SGD:** As a more commonly used optimizer, AdamW offers greater stability in parameter updates compared to SGD, effectively handling sparse gradients and non-stationary objectives, thus accelerating model convergence.
- **Incorporating a Scheduler:** The implementation of a scheduler allows for convenient control over learning rate variations.
- **Leveraging wandb for Training Monitoring:** Employing wandb for tracking the training process not only aids in monitoring model training but also facilitates automatic hyperparameter optimization. This real-time monitoring and parameter tuning can expedite the identification of the optimal model configuration.
- **Utilizing GPU Acceleration:** Training on GPUs significantly boosts training speed, enabling us to train for more epochs and swiftly experiment with various hyperparameter combinations.

2.2 Part II: Image Retrieval

Since the feature extraction process has been provided, our experiment mainly focused on the implementation of similarity computation and a top-5 sorting mechanism.

2.2.1 Similarity Computation

When provided with a vector representing the feature of an input image and a matrix where each row portrays the feature of a dataset image, both cosine similarity and Euclidean similarity can be easily implemented.

```python
def cosine_similarity(input_feature, feature_vectors):
    input_feature = input_feature / np.linalg.norm(input_feature) # input_feature.shape = (D,)
    feature_vectors = feature_vectors / np.linalg.norm(feature_vectors, axis=1, keepdims=True) # feature_vectors.shape = (N, D)
    similarities = np.dot(feature_vectors, input_feature.T) # similarities.shape = (D, 1)
    return similarities.flatten()

def euclidean_similarity(input_feature, feature_vectors):
    input_feature = input_feature / np.linalg.norm(input_feature)
    feature_vectors = feature_vectors / np.linalg.norm(feature_vectors, axis=1, keepdims=True)
    distances = np.linalg.norm(feature_vectors - input_feature, axis=1)
    similarities = 1 / (1 + distances) # Adjust similarities to [0, 1]
    return similarities
```

2.2.2 Top5 Sorting

When the dataset consists of a small number of images, sorting to identify the top 5 similarities is straightforward.

```
# similarities contains similarity values between input image and all dataset images
top5_indices = np.argsort(similarities)[::-1][:5]
top5_similarity_values = similarities[top5_indices]
top5_image_paths = image_paths[top5_indices]
```

However, in the case of a large dataset, sorting the similarity vectors can be time-consuming. In such scenarios, a priority queue can be employed for efficient sorting.

3 Experimental Results (实验结果)

3.1 Experimental Environment

The experiments were conducted on macOS Sonoma 14.6.1 with Python 3.12 and an Apple M2 Pro GPU, utilizing PyTorch 2.5.1. Additionally, a portion of the experiments was run on a server equipped with PyTorch 2.5.1, Python 3.12 (Ubuntu 22.04), CUDA 12.4, and an RTX 4090D GPU.

3.2 Classification Results

The ResNet-20 model was trained using a learning rate of 0.01, a batch size of 128, the AdamW optimizer with a weight decay of 0.01, and a StepLR scheduler with a step size of 40 and a gamma of 0.1. As Figure 1 and Figure 2 shows, after 50 epochs of training, the loss decreased to 0.108, the training accuracy improved to 96.234%, and the test accuracy reached **90.930%**.

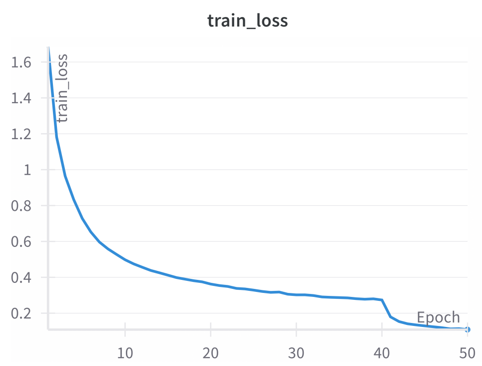

Figure 1 Loss Curve

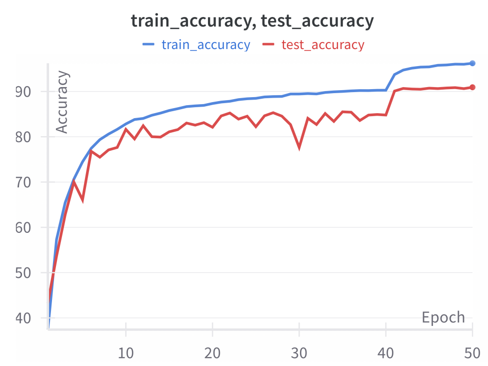

Figure 2 Training and Test Accuracy

3.3 Retrieval Results

By randomly selecting images from the 10 classes of CIFAR-10, we created a custom dataset comprising 994 images. Figure 3 showcases a selection of example images from this dataset.

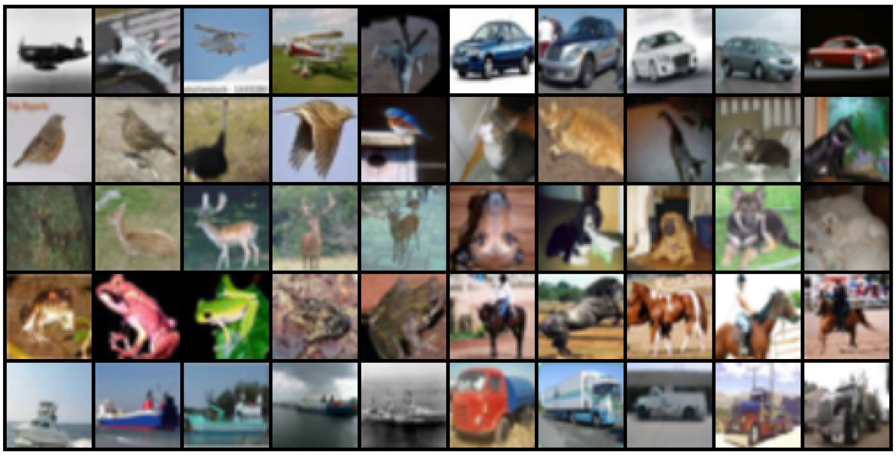

Figure 3 Example Images from the Custom Dataset

Some images not present in the custom dataset were used for retrieval purposes. The retrieval outcomes, displayed in Figure 4, exhibit a high degree of success. This is evident from the results, which effectively locate images containing similar objects.

These results highlight the capability of ResNet-50 in extracting features and emphasize the accuracy of our similarity computation methods in identifying images with similar content.

4 Analysis and Discussion (分析与思考)

4.1 Influence of Hyperparameters

Hyperparameters play a crucial role in training neural networks. In our experiment, the Sweep function within Weights & Biases was employed to identify the optimal combination of hyperparameters, focusing primarily on the selection of **learning rate and weight decay**.

4.1.1 Influence of Learning Rate

Figure 5 depicts the loss curves corresponding to different learning rates (0.1, 0.01, 0.001, 0.0001) under the AdamW optimizer, with 50 epochs, a batch size of 128, and a StepLR scheduler with a step size of 20 and a gamma of 0.1.

The graph illustrates that excessively small learning rates, such as 0.001 or 0.0001, result in slow convergence, with the training loss decreasing at a sluggish pace. Conversely, when the learning rate is too large, for instance, 0.1, overly large step sizes can cause the optimization process to overshoot and potentially miss the local minima of the loss function. Through the experiment, it was observed that **a learning rate of 0.01 strikes a balance**, facilitating efficient convergence and avoiding the pitfalls associated with rates that are either too small or too large.

4.1.2 Tuning Hyperparameters with Sweep

Utilizing the Sweep functionality, we delved into the process of fine-tuning hyperparameters to enhance the performance of our neural network model. Figure 6 presents a detailed overview of the test accuracy

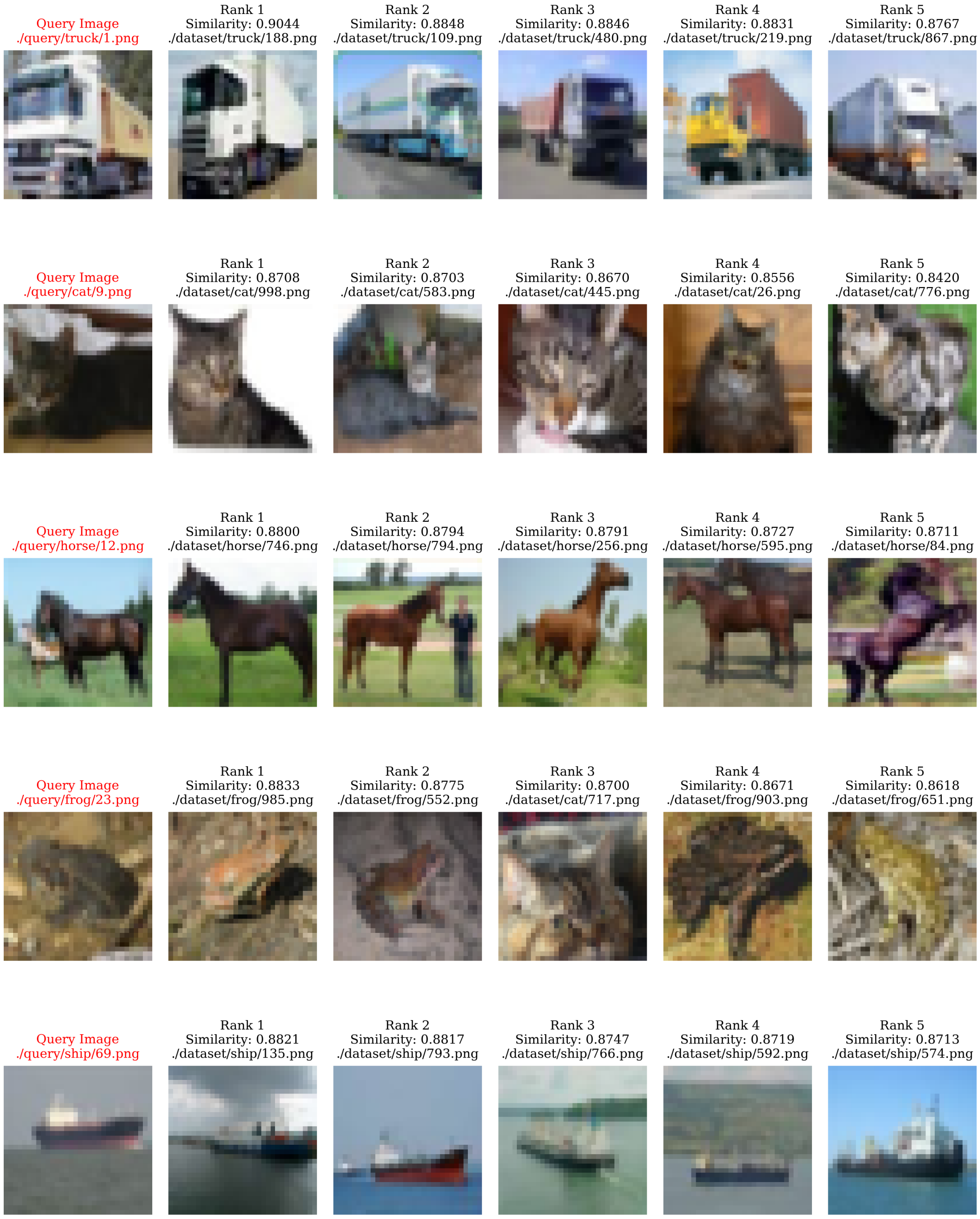

Figure 4 Retrieval Results

Figure 5 Loss Curves under Different Learning Rates

outcomes obtained through the exploration of varying learning rates (0.1, 0.01, 0.001, 0.0001) and weight decays (0.01, 0.001, 0.0001) using the Sweep feature.

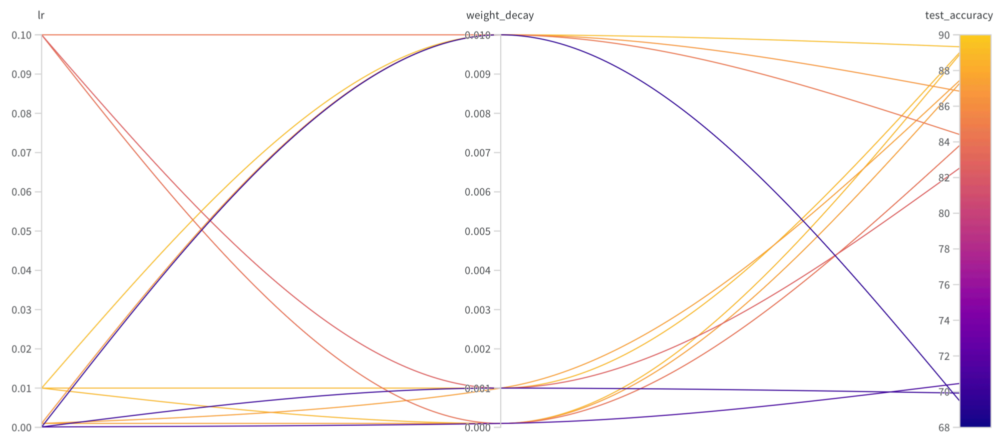

Figure 6 Test Accuracy with Different Learning Rates and Weight Decays

Notably, the experimental results highlight that **a learning rate of 0.01 coupled with a weight decay of 0.01** stands out as the optimal configuration, yielding the highest test accuracy among the evaluated combinations. This meticulous tuning process exemplifies the critical role that hyperparameter selection plays in maximizing the effectiveness of neural network training and ultimately enhancing model performance.

4.2 Problem of Overfitting

In Figure 2, a noticeable disparity is evident between the train accuracy and test accuracy, indicative of a prevalent issue in machine learning termed **overfitting**. Overfitting occurs when a model captures noise from the training data rather than learning the underlying patterns, leading to poor generalization on unseen data.

To mitigate overfitting, various regularization techniques such as weight decay, dropout, and data augmentation can be employed. In our experiment, **data augmentation** was implemented as a strategy to alleviate overfitting. Data augmentation involves artificially increasing the diversity of the training dataset by applying transformations such as cropping, flipping, color jittering, rotation, etc., to introduce variability and enhance the model's ability to generalize. The modified code snippet incorporating data augmentation is presented below:

```
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1), # Add
        color jitter
    transforms.RandomRotation(15), # Random rotation
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])
```

Figure 7 displays the train and test accuracy when employing data augmentation. A comparative analysis with Figure 2 underscores the effectiveness of data augmentation in minimizing the disparity between train accuracy and test accuracy, thereby notably mitigating the issue of overfitting.

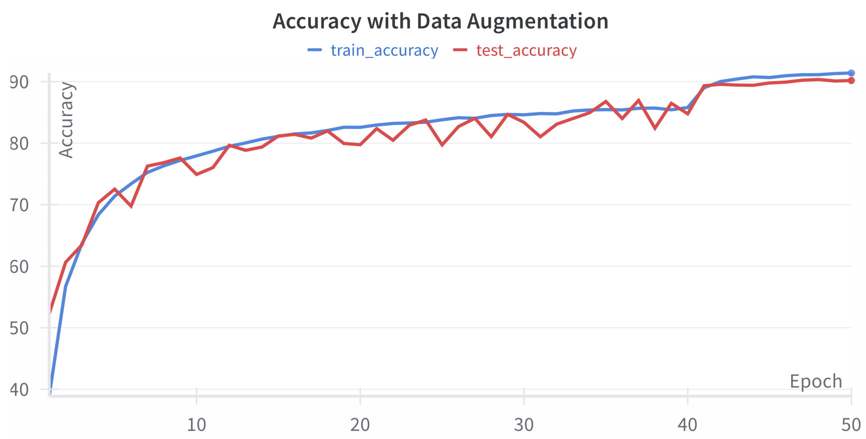

Figure 7 Train and Test Accuracy with Data Augmentation

4.3 Learning Rate Adjustment

During our training process. a StepLR scheduler was utilized, so that the learning rate decayed from 0.01 to 0.001 at the 40th epoch. Upon analyzing the trends illustrated in Figures 1 and 2, it becomes evident that initially, both the loss function and accuracy metrics were converging. However, with the subsequent decay in the learning rate, the loss exhibited a further decline while the accuracy demonstrated a notable uptick.

This observed enhancement can be attributed to the strategic adjustment of the learning rate. By lowering the learning rate at a specific epoch, the model was able to navigate the optimization landscape more effectively, fine-tuning its parameters with greater precision. This adjustment likely enabled the model to overcome local minima and plateaus that might have hindered its progress, ultimately leading to improved convergence and higher accuracy on the test data.

4.4 ViT-B/16 Comparison in Image Retrieval

The vision transformer architecture has gained prominence in recent years, excelling in various tasks including image classification and image retrieval. In our experiment, we employed ViT-B/16 to undertake the same image retrieval task as ResNet-50, enabling a comparative analysis.

Figure 8 illustrates the retrieval outcomes of both ResNet-50 and ViT-B/16. Notably, ViT-B/16 outperformed ResNet-50 in this task. The images identified by ResNet-50 as having the top 5 similarities predominantly share a red color theme, even including a red truck. Conversely, the images retrieved by ViT-B/16 all feature automobiles, with some not exhibiting a red coloration. This contrast suggests that ViT-B/16 captures more high-level features beyond color attributes.

Figure 8 Comparison of Retrieval Results between ResNet-50 (Top) and ViT-B/16 (Bottom)

5 Reflection and Conclusion (实验感想)

5.1 Learning from Experiments

Throughout these experiments, our understanding of neural networks and image retrieval has been significantly enriched. We have refined our ability to select optimal hyperparameters and further polished our proficiency in LaTeX. These experiences have not only enhanced our research capabilities but have also proven invaluable for individuals venturing into scientific exploration.

5.2 Challenges Encountered

During the course of our experiments, we encountered challenges in the **selection of hyperparameters**, which had a substantial impact on the accuracy of our models. To address this issue, we leveraged the WandB sweep functionality to conduct experiments efficiently. However, we found that this process was time-consuming. In our quest for optimal hyperparameters, we even delved into distributed computing, enabling us to run multiple networks simultaneously. This exploration allowed us to expedite the parameter tuning process.

5.3 Conclusion

In this experiment, we utilized ResNet for image classification and image retrieval tasks, yielding commendable results. Our investigation focused on strategies for hyperparameter selection, techniques to combat overfitting, and the importance of learning rate adjustments. Additionally, we conducted a comparative analysis of the retrieval performance between ResNet-50 and ViT-B/16 models.

A Source Code File List

Table 1 File List

File Name	Description
train.py	Image Classification
retrieval.py	Image Retrieval

B Source Code

```
"""
File name: train.py
"""
import os
import sys
import argparse
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import wandb
from model import resnet20


def parse_args(args):
    """
    Parse command line arguments
    """
    parser = argparse.ArgumentParser()
    parser.add_argument("--lr", type=float, default=0.01, help="Learning rate")
    parser.add_argument("--batch_size", type=int, default=128, help="Batch size")
    parser.add_argument("--epochs", type=int, default=50, help="Number of training
        epochs")
    parser.add_argument("--optimizer", type=str, default="AdamW", choices=["SGD", "Adam",
        "AdamW"], help="Optimizer type")
    parser.add_argument("--weight_decay", type=float, default=0.01, help="Weight decay")
    parser.add_argument("--step_size", type=int, default=40, help="Step size for learning
        rate scheduler")
    parser.add_argument("--augment", type=bool, default=False, help="Augment data")

    args = parser.parse_args(args)
```

```python
    return args


def main(args):
    # Get arguments
    args = parse_args(args)

    # Initialize Weights & Biases
    wandb.init(
        project="Lab5-ResNet20",
        name=f"resnet20_lr_{args.lr}_batch_{args.batch_size}_epochs_{args.epochs}_optimizer_{args.optimizer}_w
        config={
            "lr": args.lr,
            "batch_size": args.batch_size,
            "epochs": args.epochs,
            "optimizer": args.optimizer,
            "weight_decay": args.weight_decay,
            "step_size": args.step_size,
            "augment": args.augment,
        }
    )
    config = wandb.config

    # Data pre-processing
    print('==> Preparing data..')
    if config.augment: # Augment data
        transform_train = transforms.Compose([
            transforms.RandomCrop(32, padding=4),
            transforms.RandomHorizontalFlip(),
            transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
                # Add color jitter
            transforms.RandomRotation(15), # Random rotation
            transforms.ToTensor(),
            transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
        ])
    else:
        transform_train = transforms.Compose([
            transforms.RandomCrop(32, padding=4),
            transforms.RandomHorizontalFlip(),
            transforms.ToTensor(),
            transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
        ])
    transform_test = transforms.Compose([
        transforms.ToTensor(),
```

```python
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
    ])

    # Get training data
    trainset = torchvision.datasets.CIFAR10(
        root='./data', train=True, download=True, transform=transform_train)
    trainloader = torch.utils.data.DataLoader(trainset, batch_size=config.batch_size,
         shuffle=True)

    # Get testing data
    testset = torchvision.datasets.CIFAR10(
        root='./data', train=False, download=True, transform=transform_test)
    testloader = torch.utils.data.DataLoader(testset, batch_size=config.batch_size,
         shuffle=False)

    classes = ("airplane", "automobile", "bird", "cat",
            "deer", "dog", "frog", "horse", "ship", "truck")

    # Model
    print('==> Building model..')
    model = resnet20()

    # Use GPU if available
    if torch.cuda.is_available():
        device = torch.device("cuda")
    elif torch.mps.is_available():
        device = torch.device("mps")
    else:
        device = torch.device("cpu")
    model = model.to(device)

    # Loss function
    criterion = nn.CrossEntropyLoss()

    # Optimizer (AdamW by default)
    if config.optimizer == "SGD":
        optimizer = optim.SGD(model.parameters(), lr=config.lr,
             weight_decay=config.weight_decay)
    elif config.optimizer == "Adam":
        optimizer = optim.Adam(model.parameters(), lr=config.lr,
             weight_decay=config.weight_decay)
    elif config.optimizer == "AdamW":
        optimizer = optim.AdamW(model.parameters(), lr=config.lr,
             weight_decay=config.weight_decay)
```

```

# Learning rate scheduler (StepLR)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=config.step_size,
    gamma=0.1)

# Training for one epoch
def train(epoch):
    model.train()
    train_loss = 0
    correct = 0
    total = 0
    for batch_idx, (inputs, targets) in enumerate(trainloader):
        inputs, targets = inputs.to(device), targets.to(device)
        optimizer.zero_grad()

        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()

        # Calculate training accuracy
        train_loss += loss.item()
        _, predicted = outputs.max(1)
        total += targets.size(0)
        correct += predicted.eq(targets).sum().item()
        print('Epoch [%d] Batch [%d/%d] Loss: %.3f | Training Acc: %.3f%% (%d/%d)'
            % (epoch, batch_idx + 1, len(trainloader), train_loss / (batch_idx + 1),
                100. * correct / total, correct, total))

    avg_train_loss = train_loss / len(trainloader)
    train_accuracy = 100. * correct / total

    return avg_train_loss, train_accuracy

# Testing for one epoch
def test(epoch):
    print('==> Testing...')
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for batch_idx, (inputs, targets) in enumerate(testloader):
            inputs, targets = inputs.to(device), targets.to(device)
```

```
                outputs = model(inputs)

                # Calculate testing accuracy
                _, predicted = outputs.max(1)
                total += targets.size(0)
                correct += predicted.eq(targets).sum().item()

        test_accuracy = 100 * correct / total
        print(f"Epoch [{epoch}] - Test Accuracy: {test_accuracy:.3f}%")

        # Save checkpoint
        print('Saving..')
        state = {
            'net': model.state_dict(),
            'acc': test_accuracy,
            'epoch': epoch
        }
        if not os.path.isdir('checkpoint'):
            os.mkdir('checkpoint')
        torch.save(state, f'./checkpoint/ckpt_{epoch}_acc_{test_accuracy:.3f}.pth')

        return test_accuracy


    # Training and Testing loop
    for epoch in range(1, config.epochs + 1):
        avg_train_loss, train_accuracy = train(epoch)
        test_accuracy = test(epoch)

        wandb.log({
            'train_loss': avg_train_loss,
            'train_accuracy': train_accuracy,
            'test_accuracy': test_accuracy,
        })

        scheduler.step()

    wandb.finish()


if __name__ == '__main__':
    main(sys.argv[1:])
```

```
"""
```

```python
File name: retrieval.py
"""
import os
import sys
import glob
import argparse
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.models as models
from torchvision.models import ResNet50_Weights
from torchvision.models import ViT_B_16_Weights
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
from PIL import Image

# Plot settings
plt.rcParams['font.family'] = 'serif'
plt.rcParams['font.size'] = 11
plt.rcParams['axes.labelsize'] = 11
plt.rcParams['xtick.labelsize'] = 11


def parse_args(args):
    """
    Parse command line arguments
    """
    parser = argparse.ArgumentParser()
    parser.add_argument("--model", type=str, default="resnet50", choices=["resnet50",
        "vit_b_16"], help="Model name")
    parser.add_argument("--dataset-dir", type=str, default="./dataset", help="Dataset
        directory")
    parser.add_argument("--target-dir", type=str, default="./query", help="Target image
        path or directory")
    parser.add_argument("--output-dir", type=str, default="./figures/retrieval",
        help="Output directory")

    args = parser.parse_args(args)
    return args


```

```python
def get_image_paths(input_dir, extensions = ("jpg", "jpeg", "png", "bmp")):
    """
    Get image paths from the given directory
    """
    pattern = f"{input_dir}/**/*"
    img_paths = []
    for extension in extensions:
        img_paths.extend(glob.glob(f"{pattern}.{extension}", recursive=True))

    if not img_paths:
        raise FileNotFoundError(f"No images found in {input_dir}. Supported formats are:
            {', '.join(extensions)}")

    return img_paths


def extract_features(model, dataset_dir, trans, device):
    """
    Extract features from the dataset
    """
    print('===> Preparing image data..')
    dataset = ImageFolder(dataset_dir, transform=trans)
    dataloader = DataLoader(dataset, batch_size=1, shuffle=False)

    print("===> Extracting features..")
    feature_vectors = []
    image_paths = []
    with torch.no_grad():
        for inputs, labels in tqdm(dataloader, desc="Extracting features"):
            inputs = inputs.to(device)
            features = model(inputs)
            feature_vectors.append(features.cpu().numpy())
            image_paths.append(dataloader.dataset.samples[len(feature_vectors) - 1][0])

    feature_vectors = np.vstack(feature_vectors)
    image_paths = np.array(image_paths)
    return feature_vectors, image_paths


def cosine_similarity(input_feature, feature_vectors):
    """
    Calculate cosine similarity between the input feature and the feature vectors
    """
    input_feature = input_feature / np.linalg.norm(input_feature)
```

```
    feature_vectors = feature_vectors / np.linalg.norm(feature_vectors, axis=1, keepdims=True)
    similarities = np.dot(feature_vectors, input_feature.T)
    return similarities.flatten()


def euclidean_similarity(input_feature, feature_vectors):
    """
    Calculate similarity based on Euclidean distance between the input feature and the feature vectors
    """
    input_feature = input_feature / np.linalg.norm(input_feature)
    feature_vectors = feature_vectors / np.linalg.norm(feature_vectors, axis=1, keepdims=True)
    distances = np.linalg.norm(feature_vectors - input_feature, axis=1)
    similarities = 1 / (1 + distances)
    return similarities


def main(args):
    args = parse_args(args)

    # Load model
    print("==> Loading model..")
    print(f"Model: {args.model}")
    if args.model == "resnet50":
        model = models.resnet50(weights=ResNet50_Weights.IMAGENET1K_V1)
    elif args.model == "vit_b_16":
        model = models.vit_b_16(weights=ViT_B_16_Weights.IMAGENET1K_V1)
    model.fc = nn.Identity() # Remove the classification head
    model.eval()

    # Use GPU if available
    if torch.cuda.is_available():
        device = torch.device("cuda")
    elif torch.mps.is_available():
        device = torch.device("mps")
    else:
        device = torch.device("cpu")
    model = model.to(device)

    # Data pre-processing
    normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                     std=[0.229, 0.224, 0.225])
```

```
    trans = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        normalize,
    ])

    # Extract features
    feature_path = f"{args.dataset_dir}/features_{args.model}.npy"
    image_paths_path = f"{args.dataset_dir}/image_paths_{args.model}.npy"
    if not os.path.exists(feature_path) or not os.path.exists(image_paths_path):
        features, image_paths = extract_features(model, args.dataset_dir, trans, device)
        print('===> Saving features..')
        np.save(feature_path, features)
        np.save(image_paths_path, image_paths)
    else:
        print('===> Loading features..')
        features = np.load(feature_path)
        image_paths = np.load(image_paths_path)

    # Get target image paths
    if os.path.isfile(args.target_dir):
        target_paths = [args.target_dir]
    else:
        target_paths = get_image_paths(args.target_dir)

    for target_path in target_paths:
        print()
        print(f"Query Image: {target_path}")

        # Preprocess the target image
        target_img = Image.open(target_path).convert('RGB')
        target_img = trans(target_img).unsqueeze(0)
        target_img = target_img.to(device)

        # Extract features from the target image
        with torch.no_grad():
            input_feature = model(target_img).cpu().numpy()

        # Calculate cosine similarity
        similarities = cosine_similarity(input_feature, features)
        # similarities = euclidean_similarity(input_feature, features)

        # Get top-5 similar images
```

```
        top5_indices = np.argsort(similarities)[::-1][:5]
        print("Top-5 Similar Indices:", top5_indices)
        top5_similarities = similarities[top5_indices]
        top5_image_paths = image_paths[top5_indices]
        print("Top-5 Similar Images:")
        for i, (path, sim) in enumerate(zip(top5_image_paths, top5_similarities)):
            print(f"{i + 1}: {path}, Similarity: {sim:.4f}")

        # Visualize the results
        plt.figure(figsize=(10, 2.5))
        plt.subplot(1, 6, 1)
        query_img = plt.imread(target_path)
        plt.imshow(query_img)
        plt.title("Query Image\n" + target_path, fontsize=9, color='red')
        plt.axis("off")
        for i, path in enumerate(top5_image_paths):
            plt.subplot(1, 6, i + 2)
            img = plt.imread(path)
            plt.imshow(img)
            plt.title(f"Rank {i + 1}\nSimilarity: {top5_similarities[i]:.4f}\n{path}",
                fontsize=9)
            plt.axis("off")
        plt.tight_layout()
        save_path =
            f"{args.output_dir}/retrieval_{args.model}_{os.path.basename(target_path)}"
        plt.savefig(save_path, dpi=300)
        print("Results saved to:", save_path)

if __name__ == '__main__':
    main(sys.argv[1:])
```