Purdue University – ECE 60146: Homework 2 – Spring 2025 1/26/2025

Logan Anderson

Contents

Tasks

3.1 Conda Environment

I set up my environment as instructed in the assignment with no issues. The *environment.yml* file will be submitted with the code.

3.2 Comparing CIFAR10 with Custom Dataset

3.2.1 Steps of Implementing

1. Prepare CIFAR10 Dataset: I prepare CIFAR10 as in the code snippet given (but I randomize the classes that are chosen). Additionally, I altered the code so that it actually retrieves 10 elements per class as the base code did not guarantee this.

```python
def load_data(data_set=CIFAR10, class_num_filter=5, total_classes=10):
  # loads data
  transform = transforms.Compose([transforms.ToTensor()])
  data = data_set(root='./data', train=True, download=True, transform=transform)

  # filters into 5 classes
  classes = sorted(sample(range(total_classes), class_num_filter))

  # retrieve 10 elements per class
  indices = []
  class_cnt = {class_: 0 for class_ in classes}
  total_required = NUM_SAMPLES // class_num_filter
  for i, (_,label) in enumerate(data):
    if label not in classes: continue
    if class_cnt[label] < total_required:
      class_cnt[label] += 1
      indices.append(i)

  subset_data = Subset(data, indices)
  return subset_data
```

2. Create Custom Dataset: I took many pictures of a calendar I have in my apartment from several different angles. Below are some examples of images I took for this task:

FIGURE 1: SELECT CALENDAR BASE IMAGES

With minor modifications to the skeleton code, I implemented this custom dataset class shown below (I added a method to retrieve the base image given a specific index and modified the __*next*__ method to be circular using modulus). Also of note, I return the class as its corresponding base image index for ease of comparison between the base images:

```python
class Custom_Dataset(Dataset):
  def __init__(self, root, transform=None):
    self.root = root
    self.image_paths = [os.path.join(root, img) for img in os.listdir(root)]
    self.transform = transform

  def get_base_image(self, index):
    img_path = self.image_paths[index]
    transform = transforms.ToTensor()
    return transform(Image.open(img_path).convert("RGB"))

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

  def __getitem__(self, index):
    img_path = self.image_paths[index % len(self.image_paths)]
    img = Image.open(img_path).convert("RGB")
    if self.transform:
      img = self.transform(img)
    return img, index % len(self.image_paths)
```

3. Apply Data Augmentation: I next applied several different transformations to the input data which I separate between the pre-processing and then dataloader transformations:

Preprocessing

- Transformations that I perform to the base images themselves for use in the dataloader
- Purpose: Reduce redundant computation in the dataloader
- Transformations
  - *Resize*: resize the image to a fixed smaller size, ensuring all images are the same size and reducing the overall quantity of pixels to handle

```python
transforms.Resize(new_size, antialias=True) # new_size=256
```

Dataloader Transformations

- Transformations I perform upon each successive access of the given iterable
- Purpose: Transform original data in random manner to add uncertainty to inputs and effectively generate extra inputs for the data loader
- *Transformations*

- *RandomHorizontalFlip*: Flips the image horizontally with a given a probability p, allowing for the image to be recognized even if the given camera image is flipped
- *GaussianBlur*: Blurs an image according to a Gaussian distribution to add some distortion to the image, so regardless of clarity, the image can be recognized
- *RandomAffine*: Applies a random affine homography to an image, so the image can be identified even at different angles
- *ToTensor*: A simple transformation that transforms the PIL image representation into a torch tensor representation
- *Normalize*: Translates the pixels so that they are normalized around a given point for ease of use in neural network architectures

```
custom_transform = transforms.Compose([
                        transforms.RandomHorizontalFlip(p=.2),
                        transforms.GaussianBlur(3, (.3,1.5)),
                        transforms.RandomAffine(degrees=20, translate=(.1,.1), 
scale=(.9, 1.1)),
                        transforms.ToTensor(),
                        transforms.Normalize((0.5,),(0.5,))])
```

In order to generate the additional samples, similar to the skeleton code, I simply ran:

```
augmented_images = [custom_dataset[i][0] for i in range(50)]
```

4. Compare Datasets

The sizes are compared and produce the output shown to the right:

```
# compare size of datasets
print(f"Size of CIFAR10 subset: {NUM_SAMPLES} images")          # 50
print(f"Size of Custom dataset: {len(custom_dataset)} images")  # 20
```

FIGURE 2: SELECT CIFAR10 IMAGES

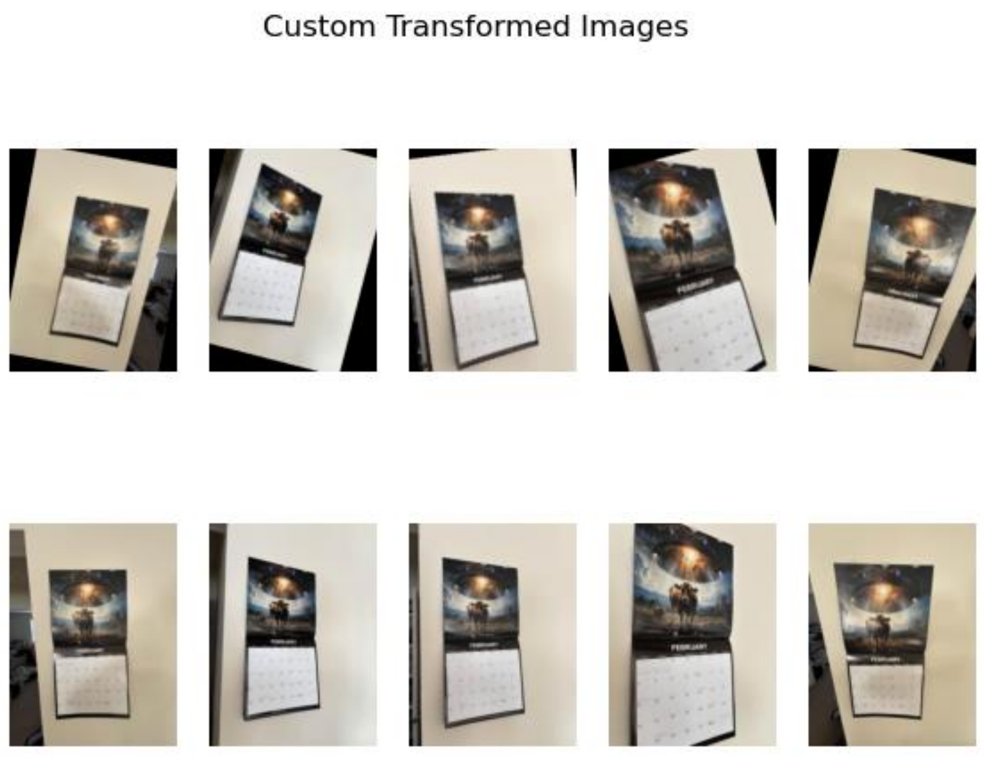

FIGURE 3: SELECT TRANSFORMED CUSTOM IMAGES (TOP) AND BASE IMAGES (BOTTOM)

3.3 Using DataLoader for Parallel Processing

I first wrap the custom dataset I created in a dataloader just as the CIFAR10 dataset was. I set $batch\_size = 4$ and $num\_workers = 2$:

```
custom_loader = DataLoader(custom_dataset, batch_size=4, num_workers=2)
```

I then plot 4 images from a single batch of the dataloader:

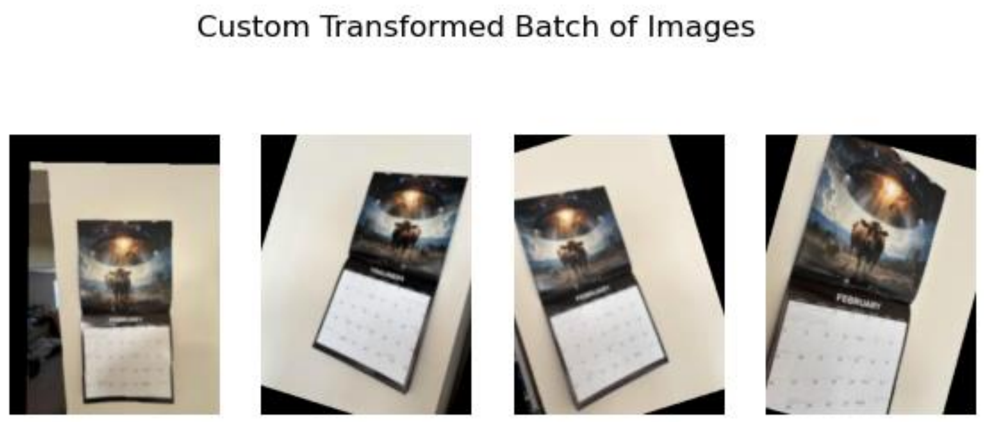

FIGURE 4: SINGLE BATCH OF CUSTOM IMAGES USING DATALOADER

Performance Comparison

First I check how long it takes to manually augment 1000 images and determine this to be around **3.579 seconds** when running the following code with $num\_images = 1000$

```
[dataset[i][0] for i in range(num_images)]
```

I then try creating 1000 images using the dataloader with varying values for *batch_size* and *num_workers*. The results of this are shown below, and I compute the average time with 5 runs each:

batch_size\num_workers	0	2	4	6
4	3.611 s	6.542 s	7.665 s	8.413 s
16	2.229 s	4.694 s	5.014 s	6.073 s
64	1.054 s	2.408 s	2.669 s	3.187 s

TABLE 1: TIMES TO GENERATE 1000 SAMPLES WITH VARYING BATCH_SIZE AND NUM_WORKERS

Next, I compare different maximum values of the actual RGB channels using the following code:

```
# get images before and after transformation
for batch_idx, (batch, labels) in enumerate(custom_loader):
  if batch_idx != 0: break
  for trans_img,label in zip(batch, labels):
    base_img = custom_dataset.get_base_image(label)
    max_vals_base = base_img.max(dim=1)[0].max(dim=1)[0]
    max_vals_trans = trans_img.max(dim=1)[0].max(dim=1)[0]

    print(f"Max values of base: {max_vals_base}")
    print(f"Max values of transformed: {max_vals_trans}")
    print()
```

The output is as follows for a batch size of 4:

```
Max values of base: tensor([1.0000, 0.9647, 0.9647])
Max values of transformed: tensor([0.9294, 0.8039, 0.8902])

Max values of base: tensor([0.9216, 0.9412, 0.9255])
Max values of transformed: tensor([0.7804, 0.8275, 0.7882])

Max values of base: tensor([0.9686, 0.9451, 0.9020])
Max values of transformed: tensor([0.9137, 0.8667, 0.7804])

Max values of base: tensor([1.0000, 0.9765, 0.9686])
Max values of transformed: tensor([0.9922, 0.9059, 0.8902])
```

3.4 Exploring Random Seed and Reproducability

The test is conducted as follows where I first run 2 times with $SEED\_SET = True$ and 2 times with $SEED\_SET = False$:

```python
custom_transform = DEFINED_CUSTOM_TRANSFORMATIONS
custom_dataset = Custom_Dataset(root=CUSTOM_OUTPUT_PATH, transform=custom_transform)

# get custom loader
custom_loader = DataLoader(custom_dataset, batch_size=2, shuffle=True)

# plot image
if SEED_SET:
  plot_data(custom_loader, path_name="custom_seed_set", img_title="Custom Seed Set Images",
num_side_by_side=2, figsize=(3,3))
else:
  plot_data(custom_loader, path_name=f"custom_seed_not_set{random.randint(0,5)}",
img_title="Custom Seed Not Set Images", num_side_by_side=2, figsize=(3,3))
```

When running the program with a seed set for 2 successive iterations, the result is as follows:

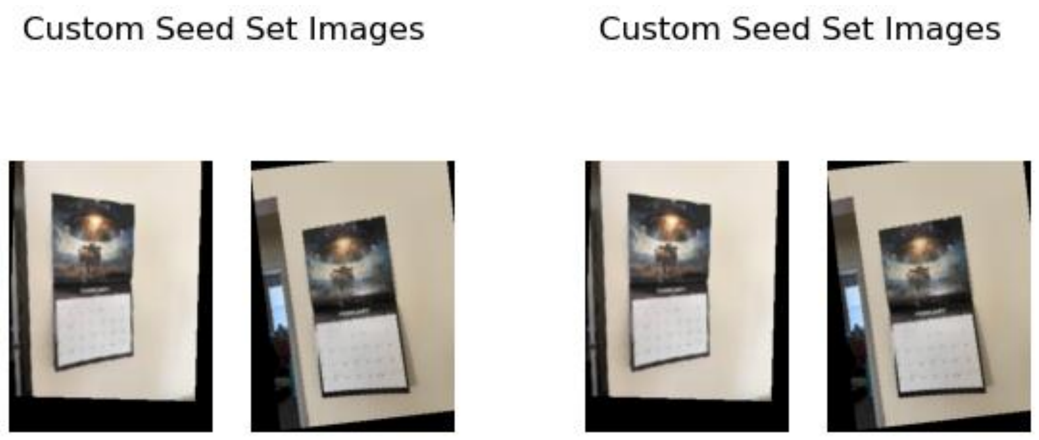

FIGURE 5: CUSTOM SEED SET IMAGES (ITERATION 1 ON THE RIGHT, ITERATION 2 ON THE LEFT)

When a seed is not set, the following is the result:

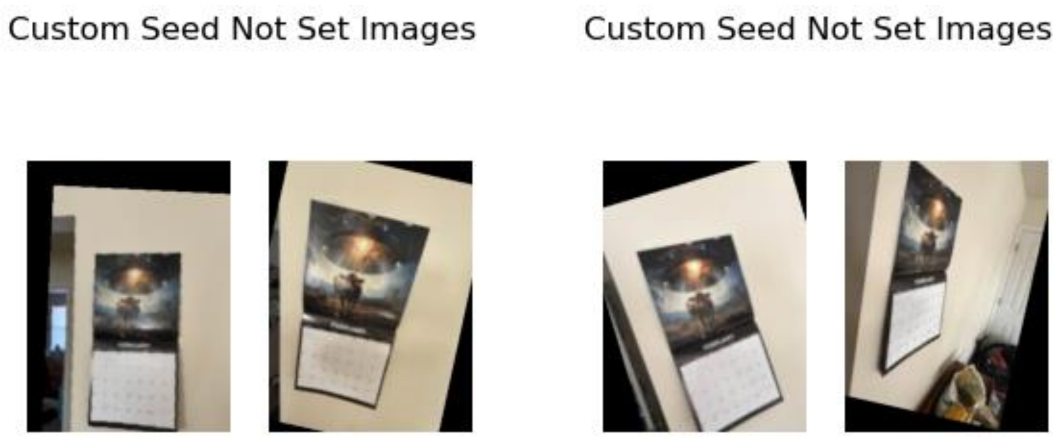

FIGURE 6: CUSTOM SEED NOT SET IMAGES (ITERATION 1 ON THE RIGHT, ITERATION 2 ON THE LEFT)

Thus, it is clear that when a seed is set, the images produced are identical, i.e. any training done will have randomness but no variation between runs. This is incredible useful when trying to test whether a given model works for predicting outputs as well as any sort of debugging in general with deep learning. When a seed isn't set, the result is different each time and thus is not as good at reproducing the same results.

Source Code

environment.yml

```
name: ece60146
channels:
  - pytorch
  - defaults
dependencies:
  - _libgcc_mutex=0.1=main
  - _openmp_mutex=5.1=1_gnu
  - blas=1.0=mkl
  - brotli-python=1.0.9=py310h6a678d5_9
  - bzip2=1.0.8=h5eee18b_6
  - ca-certificates=2024.12.31=h06a4308_0
  - certifi=2024.12.14=py310h06a4308_0
  - charset-normalizer=3.3.2=pyhd3eb1b0_0
  - cuda-cudart=12.4.127=h99ab3db_0
  - cuda-cudart_linux-64=12.4.127=hd681fbe_0
  - cuda-cupti=12.4.127=h6a678d5_1
  - cuda-libraries=12.4.1=h06a4308_1
  - cuda-nvrtc=12.4.127=h99ab3db_1
  - cuda-nvtx=12.4.127=h6a678d5_1
  - cuda-opencl=12.4.127=h6a678d5_0
  - cuda-runtime=12.4.1=hb982923_0
  - cuda-version=12.4=hbda6634_3
  - ffmpeg=4.3=hf484d3e_0
  - filelock=3.13.1=py310h06a4308_0
  - freetype=2.12.1=h4a9f257_0
  - giflib=5.2.2=h5eee18b_0
  - gmp=6.2.1=h295c915_3
  - gmpy2=2.1.2=py310heeb90bb_0
  - gnutls=3.6.15=he1e5248_0
  - idna=3.7=py310h06a4308_0
  - intel-openmp=2023.1.0=hdb19cb5_46306
  - jinja2=3.1.4=py310h06a4308_1
  - jpeg=9e=h5eee18b_3
  - lame=3.100=h7b6447c_0
  - lcms2=2.16=hb9589c4_0
  - ld_impl_linux-64=2.40=h12ee557_0
  - lerc=4.0.0=h6a678d5_0
  - libcublas=12.4.5.8=h99ab3db_1
  - libcufft=11.2.1.3=h99ab3db_1
  - libcufile=1.9.1.3=h99ab3db_1
  - libcurand=10.3.5.147=h99ab3db_1
  - libcusolver=11.6.1.9=h99ab3db_1
  - libcusparse=12.3.1.170=h99ab3db_1
  - libdeflate=1.22=h5eee18b_0
  - libffi=3.4.4=h6a678d5_1
  - libgcc-ng=11.2.0=h1234567_1
  - libgomp=11.2.0=h1234567_1
  - libiconv=1.16=h5eee18b_3
  - libidn2=2.3.4=h5eee18b_0
  - libjpeg-turbo=2.0.0=h9bf148f_0
  - libnpp=12.2.5.30=h99ab3db_1
  - libnvfatbin=12.4.127=h7934f7d_2
  - libnvjitlink=12.4.127=h99ab3db_1
  - libnvjpeg=12.3.1.117=h6a678d5_1
  - libpng=1.6.39=h5eee18b_0
  - libstdcxx-ng=11.2.0=h1234567_1
  - libtasn1=4.19.0=h5eee18b_0
  - libtiff=4.5.1=hffd6297_1
  - libunistring=0.9.10=h27cfd23_0
  - libuuid=1.41.5=h5eee18b_0
  - libwebp=1.3.2=h11a3e52_0
  - libwebp-base=1.3.2=h5eee18b_1
  - llvm-openmp=14.0.6=h9e868ea_0
  - lz4-c=1.9.4=h6a678d5_1
  - markupsafe=2.1.3=py310h5eee18b_1
  - mkl=2023.1.0=h213fc3f_46344
  - mkl-service=2.4.0=py310h5eee18b_2
  - mkl_fft=1.3.11=py310h5eee18b_0
  - mkl_random=1.2.8=py310h1128e8f_0
  - mpc=1.1.0=h10f8cd9_1
  - mpfr=4.0.2=hb69a4c5_1
  - mpmath=1.3.0=py310h06a4308_0
  - ncurses=6.4=h6a678d5_0
  - nettle=3.7.3=hbbd107a_1
  - networkx=3.4.2=py310h06a4308_0
  - numpy=1.26.4=py310h5f9d8c6_0
  - numpy-base=1.26.4=py310hb5e798b_0
  - ocl-icd=2.3.2=h5eee18b_1
  - openh264=2.1.1=h4ff587b_0
  - openjpeg=2.5.2=he7f1fd0_0
  - openssl=3.0.15=h5eee18b_0
  - pillow=11.0.0=py310hcea889d_1
  - pip=24.2=py310h06a4308_0
  - pysocks=1.7.1=py310h06a4308_0
  - python=3.10.16=he870216_1
  - pytorch=2.5.1=py3.10_cuda12.4_cudnn9.1.0_0
  - pytorch-cuda=12.4=hc786d27_7
  - pytorch-mutex=1.0=cuda
  - pyyaml=6.0.2=py310h5eee18b_0
  - readline=8.2=h5eee18b_0
  - requests=2.32.3=py310h06a4308_1
  - setuptools=75.1.0=py310h06a4308_0
  - sqlite=3.45.3=h5eee18b_0
  - sympy=1.13.3=py310h06a4308_0
  - tbb=2021.8.0=hdb19cb5_0
  - tk=8.6.14=h39e8969_0
  - torchtriton=3.1.0=py310
  - torchvision=0.20.1=py310_cu124
  - typing_extensions=4.12.2=py310h06a4308_0
  - tzdata=2025a=h04d1e81_0
  - urllib3=2.3.0=py310h06a4308_0
  - wheel=0.44.0=py310h06a4308_0
  - xz=5.4.6=h5eee18b_1
  - yaml=0.2.5=h7b6447c_0
  - zlib=1.2.13=h5eee18b_1
  - zstd=1.5.6=hc292b87_0
prefix: /home/the_linux_ruler/anaconda3/envs/ece60146

```

hw2.py

```python
import torch
import torch.backends
import torch.backends.cudnn
from torchvision import transforms
from torchvision.datasets import CIFAR10
from torch.utils.data import DataLoader, Subset, Dataset
from PIL import Image
from random import sample
import random
import os
import matplotlib.pyplot as plt
from math import ceil
from time import time
import inspect
import csv
import numpy as np

NUM_SAMPLES = 50
NUM_SAMPLES_PARALLEL = 1000
NUM_ITERATIONS_PER_TEST = 5
NUM_WORKERS = [0,2,4,6]
BATCH_SIZES = [4,16,64]
CUSTOM_IMAGE_SIZE = 256
DEFINED_CUSTOM_TRANSFORMATIONS = transforms.Compose([
                                                      # transforms.Resize(CUSTOM_IMAGE_SIZE, antialias=True),
                                                      transforms.RandomHorizontalFlip(p=.2),
                                                      transforms.GaussianBlur(3, (.3,1.5)),
                                                      transforms.RandomAffine(degrees=20, translate=(.1,.1), scale=(.9, 1.1)),
                                                      transforms.ToTensor(),
                                                      transforms.Normalize((0.5,),(0.5,))])
RESULTS_PATH = '/home/the_linux_ruler/ECE60146/HW2/results/'
CUSTOM_INPUT_PATH = "/home/the_linux_ruler/ECE60146/HW2/custom_images"
```

```python
CUSTOM_OUTPUT_PATH = "/home/the_linux_ruler/ECE60146/HW2/custom_images2"
RUN_SEED_TEST = True
SEED_SET = False
DEFINED_SEED = 60146


# -----------------------------------------------------------------------------------------
def current_time():
  return round(time() - start_time, 3)

def current_time_str():
  curr_time = current_time()
  if curr_time // 3600:
    return f"{int(curr_time) // 3600} hr\t{(int(curr_time) % 3600) // 60} min\t{round((curr_time - int(curr_time)) + (int(curr_time) % 60),3)} sec"
  elif curr_time // 60:
    return f"{int(curr_time) // 60} min\t {round((curr_time - int(curr_time)) + (int(curr_time) % 60), 3)} sec"
  else:
    return f"{curr_time} sec"

def time_diff(time_start):
  return round(time() - time_start, 3)

def time_func(func, *args, print_params=[], print_extra="", get_time=False, suppress_output=False):
  time_start = time()
  params_str = ",".join([f"{names_in_caller(param, depth=3)[0]} = {param}" for param in print_params])
  extra_string = f" with parameter(s) {params_str}" if len(print_params) != 0 else ""
  extra_string += " " + print_extra
  if not suppress_output: print(f"{func.__name__}{extra_string} starts at time {current_time_str()}")
  ret_val = func(*args)
  time_taken = time_diff(time_start)
  if not suppress_output: print(f"{func.__name__}{extra_string} ends at time {current_time_str()} in {time_taken} seconds")
  if get_time: return ret_val, time_taken
  return ret_val

def namestr(obj, namespace):
  return [name for name in namespace if namespace[name] is obj]

def names_in_caller(obj, depth=2) -> list[str]:
  f = inspect.currentframe()
  for _ in range(depth): f = f.f_back
  return namestr(obj, f.f_locals)

def resize_image(input_path, output_path, new_size):
  # Open the image file
  img = Image.open(input_path)

  resize_transform = transforms.Compose([
      transforms.Resize(new_size, antialias=True),  # Resize image
      transforms.ToTensor()  # Convert to tensor
  ])

  resized_img = resize_transform(img)
  resized_img = transforms.ToPILImage()(resized_img)

  # Save resized image
  resized_img.save(output_path)

def set_seed(seed_num):
  random.seed(seed_num)
  torch.manual_seed(seed_num)
  torch.cuda.manual_seed(seed_num)
  np.random.seed(seed_num)
  torch.backends.cudnn.deterministic = True
  torch.backends.cudnn.benchmark = False
  os.environ['PYTHONHASHSEED'] = str(seed_num)
# ------------------------------------------------------------------------------------------

class Custom_Dataset(Dataset):
  def __init__(self, root, transform=None):
    self.root = root
    self.image_paths = [os.path.join(root, img) for img in os.listdir(root)]
    self.transform = transform

  def get_base_image(self, index):
    img_path = self.image_paths[index]
    transform = transforms.ToTensor()
    return transform(Image.open(img_path).convert("RGB"))

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

  def __getitem__(self, index):
    img_path = self.image_paths[index % len(self.image_paths)]
    img = Image.open(img_path).convert("RGB")
    if self.transform:
      img = self.transform(img)
    return img, index % len(self.image_paths)

def load_data(data_set=CIFAR10, class_num_filter=5, total_classes=10):
  # loads data
  transform = transforms.Compose([transforms.ToTensor()])
  data = data_set(root='./data', train=True, download=True, transform=transform)

  # filters into 5 classes
  classes = sorted(sample(range(total_classes), class_num_filter))

  # retrieve 10 elements per class
  indices = []
  class_cnt = {class_: 0 for class_ in classes}
  total_required = NUM_SAMPLES // class_num_filter
  for i, (_,label) in enumerate(data):
    if label not in classes: continue
    if class_cnt[label] < total_required:
      class_cnt[label] += 1
      indices.append(i)

  subset_data = Subset(data, indices)
  return subset_data

def transform_back_to_rgb(arr):
  arr = np.array(arr)
  min_old = np.min(arr)
  max_old = np.max(arr)

  arr = ((arr - min_old) / (max_old - min_old)) * (255 - 0) + 0
  return arr.astype(int)


def plot_data(data_loader:DataLoader, path_name:str, img_title:str, num_side_by_side=5,
total_batches=1, figsize=None, plot_base_image=False, dataset=None):
  num_up_down = ceil(data_loader.batch_size * total_batches / num_side_by_side)

  if plot_base_image: num_up_down *= 2

  # configures figure size if specified
  if figsize == None: fig, axs = plt.subplots(num_up_down, num_side_by_side)
  else: fig, axs = plt.subplots(num_up_down, num_side_by_side, figsize=figsize)

  # adjusts axs to work for 1-D array
  if num_up_down == 1: axs = [axs]

  targets = []
  for i,(batch, target) in enumerate(data_loader):
    offset = i * data_loader.batch_size
    if i >= total_batches: break
    targets.append(target)
    for j,img in enumerate(batch):
      y,x = (offset + j) // num_side_by_side, (offset + j) % num_side_by_side
      # if num_up_down == 1: x,y = y,x
      axs[y][x].imshow(transform_back_to_rgb(img.permute(1,2,0).numpy()))
      axs[y][x].axis("off")

  if plot_base_image:
    for i,target in enumerate(targets):
      for j,target_scalar in enumerate(target):
        base_img = dataset.get_base_image(target_scalar)
        loc = offset + i + j
        y,x = loc // num_side_by_side, loc % num_side_by_side
        axs[y][x].imshow(transform_back_to_rgb(base_img.permute(1,2,0).numpy()))
        axs[y][x].axis("off")

  plt.suptitle(img_title)

  fig.savefig(f"{RESULTS_PATH}{path_name}")

def augment_images(dataset, num_images):
  return [dataset[i][0] for i in range(num_images)]

def get_images_parallel(data_loader:DataLoader, num_images:int):
  imgs = []
  iter_data = iter(data_loader)
  for _ in range(num_images // data_loader.batch_size):
    try:
      imgs.append(next(iter_data))
    except StopIteration:
      iter_data = iter(data_loader)
      imgs.append(next(iter_data))

  return imgs

def create_csv(filename, headers, data):
  with open(f"{RESULTS_PATH}{filename}.csv", mode='w', newline='', encoding='utf-8') as file:
    writer = csv.writer(file)
    writer.writerow(headers)
    for data_point in data:
      writer.writerow(data_point)

if __name__ == "__main__":
  start_time = time()

  # set seed for same result (for testing purposes)
  if SEED_SET: set_seed(DEFINED_SEED)
  if RUN_SEED_TEST:
    custom_transform = DEFINED_CUSTOM_TRANSFORMATIONS
    custom_dataset = Custom_Dataset(root=CUSTOM_OUTPUT_PATH, transform=custom_transform)

    # get custom loader
    custom_loader = DataLoader(custom_dataset, batch_size=2, shuffle=True)

    # plot image
    if SEED_SET:
      plot_data(custom_loader, path_name="custom_seed_set", img_title="Custom Seed Set Images", num_side_by_side=2, figsize=(3,3))
    else:
      plot_data(custom_loader, path_name=f"custom_seed_not_set{random.randint(0,5)}", img_title="Custom Seed Not Set Images", num_side_by_side=2, figsize=(3,3))

    exit()

  data = load_data()

  # resize custom images
  print("Resizing Images")
  for filename in os.listdir(CUSTOM_INPUT_PATH):
    resize_image(os.path.join(CUSTOM_INPUT_PATH, filename), os.path.join(CUSTOM_OUTPUT_PATH, filename), CUSTOM_IMAGE_SIZE)

  # create custom transformation and apply with custom dataset
  custom_transform = DEFINED_CUSTOM_TRANSFORMATIONS
  custom_dataset = Custom_Dataset(root=CUSTOM_OUTPUT_PATH, transform=custom_transform)
  augmented_images = augment_images(custom_dataset, NUM_SAMPLES)

  # compare size of datasets
  print(f"Size of CIFAR10 subset: {NUM_SAMPLES} images")
  print(f"Size of Custom dataset: {len(custom_dataset)} images")

  # load subset of CIFAR10
  cifar_loader = DataLoader(data, batch_size=5, shuffle=False)

  # load custom dataset into dataloader
  custom_loader = DataLoader(custom_dataset, batch_size=4, num_workers=2)

  # get images before and after transformation
  for batch_idx, (batch, labels) in enumerate(custom_loader):
    if batch_idx != 0: break
    for trans_img,label in zip(batch, labels):
      base_img = custom_dataset.get_base_image(label)
      max_vals_base = base_img.max(dim=1)[0].max(dim=1)[0]
      max_vals_trans = trans_img.max(dim=1)[0].max(dim=1)[0]

      print(f"Max values of base: {max_vals_base}")
      print(f"Max values of transformed: {max_vals_trans}")
      print()

  # create custom loader with a batch size of 5
  custom_loader2 = DataLoader(custom_dataset, batch_size=5)

  # plot various examples
  plot_data(cifar_loader, "CIFAR10_Examples", "CIFAR10 Subset Samples", figsize=(7,3))
  plot_data(custom_loader2, path_name="custom_examples", img_title="Custom Transformed Images", num_side_by_side=5, figsize=(7,5), plot_base_image=True, dataset=custom_dataset)
  plot_data(custom_loader, path_name="custom_example_batch", img_title="Custom Transformed Batch of Images", num_side_by_side=4, figsize=(7,3))

  # augment 1000 images and calculate how long this takes
  _, time_taken = time_func(augment_images, custom_dataset, NUM_SAMPLES_PARALLEL, get_time=True)

  # compute times for different batch sizes and num_workers (1000)
  all_points = []
```

```
for batch_size in BATCH_SIZES:
  row = []
  row.append(batch_size)
  for num_workers in NUM_WORKERS:
    print(f"Running calculation for batch_size={batch_size}, num_workers={num_workers} at time {current_time_str()}")
    times_taken = []
    for _ in range(NUM_ITERATIONS_PER_TEST):
      custom_loader = DataLoader(custom_dataset, batch_size=batch_size, num_workers=num_workers)
      _, time_taken = time_func(get_images_parallel, custom_loader, NUM_SAMPLES_PARALLEL, print_extra=f"with num_workers={num_workers}, batch_size={batch_size}", get_time=True, suppress_output=True)
      times_taken.append(time_taken)
    row.append(sum(times_taken) / len(times_taken))
  all_points.append(row)

# generate csv file to store this information
create_csv("time_comparisons2", ["batch_size\\num_workers", *NUM_WORKERS], all_points)
```