ECE 60146 and BME 64600: Homework 2

Triparna Mahata

January 2026

1 Introduction

Goal of this homework:

- To set up a dedicated local development environment using Anaconda and install essential deep learning libraries, specifically PyTorch.
- To gain practical experience with fundamental image handling techniques in a deep learning context, with specific focus on image scaling and normalization concepts.
- To apply various image transformations to demonstrate a deeper understanding of data augmentation strategies and their utility in training models.

2 Data Augmentation and Transformations

Data augmentation is a powerful technique to artificially expand the size and diversity of a training dataset by applying random transformations to images. This helps reduce overfitting and improves the model's ability to generalize to new, unseen data. In PyTorch, augmentations are performed using the `torchvision.transforms` module.

2.1 Common Data Augmentation Techniques

Data augmentation is typically applied to the images after pixel-value scaling and pixel-value normalization. We will apply the following transformations for pixel value scaling and normalization:

- `ToTensor`: Converts PIL Image to PyTorch tensor and scales pixel values from [0, 255] to [0.0, 1.0].
- `Normalize`: Normalizes tensor values using channel-wise mean and standard deviation.

The following transformations are commonly used for image augmentation:

- `RandomHorizontalFlip`: Randomly flips the image horizontally with a given probability.
- `GaussianBlur`: Applies Gaussian blur with a specified kernel size and sigma range.
- `RandomAffine`: Applies random affine transformations including rotation, translation, and scaling.

A typical transformation pipeline typically combines all of the transformations listed above through a callable instance of `torchvision.transforms.Compose`

2.2 PyTorch Dataset and DataLoader

PyTorch provides the `Dataset` class for representing datasets. The `DataLoader` class works alongside `Dataset` to handle batching, shuffling and parallel loading of data.

3 Programming Tasks

```
"""
ECE 60146 - Homework 2
Name: Triparna Mahata
Email: tmahata@purdue.edu

ImageNet Data Loading and Augmentation
"""

import os
import torch
import numpy as np
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from PIL import Image
import time
import multiprocessing
multiprocessing.set_start_method('spawn', force=True)
```

3.1 Conda Environment

`environment.yml` file submitted as part of the zip file `hw2_TriparnaMahata.zip`.

3.2 Working with ImageNet

I obtained ImageNet images using the 'Option A': Downloaded the pre-prepared ImageNet subset from Brightspace.

```
REQUIRED_CLASSES = {
    1: "goldfish",
    151: "Chihuahua",
    281: "tabby_cat",
    291: "lion",
    325: "sulphur_butterfly",
    386: "African_elephant",
    430: "basketball",
    466: "bullet_train",
    496: "Christmas_stocking",
    950: "orange"
}

def get_class_name(label):
    """Get class name from integer label."""
    return REQUIRED_CLASSES.get(label, f"class_{label}")

# TRANSFORMS
transform_custom = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.2),
    transforms.RandomAffine(degrees=20, translate=(0.1,0.1), scale=(0.9,1.1)),
    transforms.Resize((224,224)),
    transforms.GaussianBlur(3,(0.3,1.5)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225])
])

transform_basic = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor()
])

# OPTION A : Dataset class for provided subset
class ImageNetSubset(Dataset):
    """
    Dataset for loading the provided ImageNet subset.
```

```
    imagenet_subset/
                  1/    (goldfish)
                    00001.JPEG
                    ...
                151/    (Chihuahua)
                    00001.JPEG
                    ...
                ...
    Args:
        root(str): Path to imagenet_subset folder
        class_labels(list): List of integer labels to load (e.g. [1, 151, 281])
        images_per_class(int): Number of images to load per class
        transform (callable): Transform to apply to images
    """
    def __init__(self, root, class_labels, images_per_class = 5, transform = None):
        self.root = root
        self.class_labels = class_labels
        self.images_per_class = images_per_class
        self.transform = transform

        self.samples = []   # List of (image_path, label)
        self._load_samples()

        print (f"Loaded {len(self.samples)} images from {len(class_labels)} classes")

    def _load_samples(self):
        """Load image paths for each requested class."""
        # For each label in self.class_labels:
        #   1. Build path to class folder: os.path.join(self.root, str(label))
        #   2. List all .JPEG /.jpg /.png files in that folder
        #   3. Take first self.images_per_class images
        #   4. Append (image_path, label) to self.samples

        for label in self.class_labels:
            class_dir = os.path.join(self.root, str(label))
            all_files = sorted(os.listdir(class_dir))
            images = [f for f in all_files if f.lower().endswith(('.jpeg', '.jpg', '.png'))]
            selected_images = images[:self.images_per_class]
            for img_name in selected_images:
                img_path = os.path.join(class_dir, img_name)
                self.samples.append((img_path, label))

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

    def __getitem__(self, index):
        """
        Returns:
            image(Tensor): Transformed image
            label(int): Class label (e.g., 1 for goldfish)
        """
        # 1. Get image_path, label from self.samples[index]
        # 2. Load image using PIL: Image.open(path).convert('RGB')
        # 3. Apply self.transform if not None
        # 4. Return (image, label)

        image_path, label = self.samples[index]
        image = Image.open(image_path).convert('RGB')
        if self.transform:
            image = self.transform(image)
        return image, label
```

3.3 Using DataLoader for Parallel Processing

1. Wrap your custom dataset class within a DataLoader instance. Set the batch size to 4 and enable multi-threading by setting `num_workers` to a value greater than 1.

```
# Custom dataset for my own images
class CustomDataset(Dataset):
    """Dataset for loading custom images from a folder."""
    def __init__(self, root, transform = None):
        self.root = root
        self.transform = transform

        # Load all image paths from root folder
        # Filter for: .jpg, .jpeg, .png, .bmp, .gif
        self.image_paths = []

        valid_extns = ('.jpg', '.jpeg', '.png', '.bmp', '.gif')

        for filename in sorted(os.listdir(self.root)):
            if filename.lower().endswith(valid_extns):
                full_path = os.path.join(self.root, filename)
                self.image_paths.append(full_path)

        print(f"Found {len(self.image_paths)} images in {root}")

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

    def __getitem__(self, index):
        img_path = self.image_paths[index]
        image = Image.open(img_path).convert('RGB')
        if self.transform:
            image = self.transform(image)
        return image, img_path
############################################################
# Utility Functions
def denormalize(tensor, mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225]):
    """ Denormalize a tensor image for display."""
    mean = torch.tensor(mean).view(3,1,1)
    std = torch.tensor(std).view(3,1,1)
    return tensor * std + mean

def show_images(images, titles, rows, cols, figsize=(15,10), save_path=None):
    """Display a grid of images."""
    fig, axes = plt.subplots(rows, cols, figsize = figsize)
    axes = axes.flatten() if rows*cols > 1 else [axes]

    for i,(img,title) in enumerate(zip(images,titles)):
        if i >= len(axes):
            break
        if isinstance(img, torch.Tensor):
            if img.min() < 0 or img.max() > 1:
                img = denormalize(img)
            img = img.permute(1,2,0).numpy()

        img = np.clip(img,0,1)
        axes[i].imshow(img)
        axes[i].set_title(title,fontsize = 10)
        axes[i].axis('off')

    plt.tight_layout()
    if save_path:
        plt.savefig(save_path, dpi=150, bbox_inches='tight')
    plt.show()
############################################################
# Utility Class
```

```
class RepeatedDataset(Dataset):
    def __init__(self, dataset, target_length=1000):
        self.dataset = dataset
        self.target_length = target_length
        self.original_length = len(dataset)

    def __len__(self):
        return self.target_length

    def __getitem__(self, index):
        return self.dataset[index % self.original_length]
############################################################
def main():
    custom_root = './my_images'
    custom_dataset = CustomDataset(root=custom_root, transform=transform_custom)
    loader = DataLoader(custom_dataset,
                        batch_size = 4,
                        shuffle = True,
                        num_workers = 2)
############################################################
if __name__ == "__main__":
    main()

```

2. Plot all 4 images from a single batch as returned by the DataLoader.

```
def main():
    custom_root = './my_images'
    custom_dataset = CustomDataset(root=custom_root, transform=transform_custom)
    loader = DataLoader(custom_dataset,
                        batch_size = 4,
                        shuffle = True,
                        num_workers = 2)

    # Plot all 4 images from a single batch as returned by the DataLoader.
    dataiter = iter(loader)
    images, paths = next(dataiter)
    titles = [os.path.basename(p) for p in paths]
    show_images(images, titles, rows=1, cols=4, save_path='batch_visualization.png')
############################################################
if __name__ == "__main__":
    main()

```

3. Compare performance for parallel data loading:

   - Generate 1000 custom images.

```
def main():
    custom_root = './my_images'
    custom_dataset = CustomDataset(root=custom_root, transform=transform_custom)
```

```
    # Generate 1000 custom images
    augmented_dataset = RepeatedDataset(custom_dataset, target_length=1000)
############################################################
if __name__ == "__main__":
    main()
```

- Measure the time taken to load and augment all 1000 images using `__getitem__` in a loop.

```
def main():
    custom_root = './my_images'
    custom_dataset = CustomDataset(root=custom_root, transform=transform_custom)

    # Generate 1000 custom images
    augmented_dataset = RepeatedDataset(custom_dataset, target_length=1000)

    # Measure time taken to load and augment all 1000 images using __getitem__ in a loop
    print(f"Time taken to load and augment all 1000 images using __getitem__ in a loop")
    start_time = time.time()
    for i in range(1000):
        _ = augmented_dataset[i]
    end_time = time.time()
    get_item_duration = end_time - start_time
    print(f"__getitem__ Loop Time: {get_item_duration:.4f} seconds")
############################################################
if __name__ == "__main__":
    main()
```

- Measure the time taken by the `DataLoader` to process all 1000 images with varying `batch_size` and `num_workers`.

```
def main():
    custom_root = './my_images'
    custom_dataset = CustomDataset(root=custom_root, transform=transform_custom)

    # Generate 1000 custom images
    augmented_dataset = RepeatedDataset(custom_dataset, target_length=1000)

    # Measure time taken by the DataLoader to process all 1000 images with varying batch
     size and num workers.
    configs = [(16, 0),
               (16, 2),
               (16, 4),
               (32, 4),
               (64, 4)]

    print(f"{'batch_size':<15} {'num_workers':<15} {'Time (sec)':<15}")
    print("-" * 45)

    for batch_size, num_workers in configs:
        loader = DataLoader(augmented_dataset,
                            batch_size = batch_size,
                            shuffle = True,
                            num_workers = num_workers)
        start_t = time.time()
        for batch_images, batch_paths in loader:
            pass
        end_t = time.time()
        duration = end_t - start_t
        print(f"{batch_size:<15} {num_workers:<15} {duration:.4f}")
############################################################
if __name__ == "__main__":
    main()
```

- Report your findings in a table.

```
Time taken to load and augment all 1000 images using __getitem__ in a loop
__getitem__ Loop Time: 12.9741 seconds
batch_size      num_workers     Time (sec)
-----------------------------------------------
16              0               12.6696
16              2               17.8498
16              4               25.1295
32              4               25.2466
64              4               24.8814

```

The multiprocessing start method 'spawn' creates fresh process instances. This introduces significant overhead because each worker process must re-import libraries and re-initialize the dataset environment every time a worker is spun up. For small datasets or simple transforms, this initialization cost outweighs the benefits of parallel loading, making `num_workers=0` significantly faster.

4. Compute the maximum of each RGB channel value of the images before and after normalization for one batch.

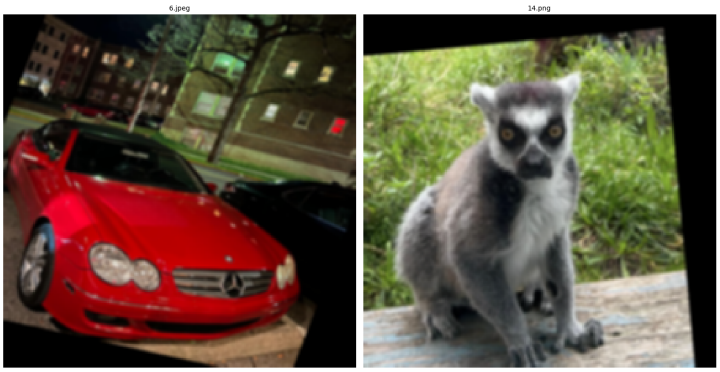
```
    def main():
    custom_root = './my_images'
    custom_dataset = CustomDataset(root=custom_root, transform=transform_custom)

    # Generate 1000 custom images
    augmented_dataset = RepeatedDataset(custom_dataset, target_length=1000)

    # Computing the maximum of each RGB channel value of the images before and after
    normalization for one batch.
    check_loader = DataLoader(augmented_dataset, batch_size=32, shuffle=True)
    images_norm, _ = next(iter(check_loader)) # These are AFTER normalization
    images_unnorm = denormalize(images_norm)

    def get_channel_max(tensor):
        # Tensor shape: [Batch, 3, Height, Width]
        # Permute to [3, Batch, Height, Width] then flatten to [3, -1]
        # This groups all R pixels, all G pixels, all B pixels
        flattened = tensor.permute(1, 0, 2, 3).reshape(3, -1)
        return flattened.max(dim=1).values

    max_vals_unnorm = get_channel_max(images_unnorm)
    max_vals_norm = get_channel_max(images_norm)

    print(f"{'Channel':<10} | {'Max (Before Norm)':<20} | {'Max (After Norm)':<20}")
    print("-" * 55)

    channel_names = ['R', 'G', 'B']
    for i, name in enumerate(channel_names):
        print(f"{name:<10} | {max_vals_unnorm[i]:.4f}{' ':>14} | {max_vals_norm[i]:.4f}")

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

```

```
Channel    | Max (Before Norm)    | Max (After Norm)
-------------------------------------------------------
R          | 1.0000               | 2.2489
G          | 1.0000               | 2.4286
B          | 1.0000               | 2.6400

```

3.4 Exploring Random Seed and Reproducibility

1. Without setting a random seed:

```python
def main():

    custom_root = './my_images'
    custom_dataset = CustomDataset(root=custom_root, transform=transform_custom)
    loader = DataLoader(custom_dataset, batch_size=2, shuffle=True)

    # --- RUN 1 ---
    data_iter = iter(loader)
    images_1, paths_1 = next(data_iter)
    titles_1 = [os.path.basename(p) for p in paths_1]
    print(f"Batch 1 contains: {titles_1}")
    show_images(images_1, titles_1, rows=1, cols=2)

    # --- RUN 2 ---
    data_iter = iter(loader)
    images_2, paths_2 = next(data_iter)
    titles_2 = [os.path.basename(p) for p in paths_2]
    print(f"Batch 2 contains: {titles_2}")
    show_images(images_2, titles_2, rows=1, cols=2)

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

```



2. With a random seed:

```python
def set_seed(seed = 60146):
    """Set random seed for reproducibility."""
    import random
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
```

```
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)

```

```
def main():

    custom_root = './my_images'
    custom_dataset = CustomDataset(root=custom_root, transform=transform_custom)
    loader = DataLoader(custom_dataset, batch_size=2, shuffle=True)

    # --- RUN 1 ---
    set_seed(60146)
    data_iter = iter(loader)
    images_1, paths_1 = next(data_iter)
    titles_1 = [os.path.basename(p) for p in paths_1]
    print(f"Batch 1 contains: {titles_1}")
    show_images(images_1, titles_1, rows=1, cols=2)

    # --- RUN 2 ---
    set_seed(60146)
    data_iter = iter(loader)
    images_2, paths_2 = next(data_iter)
    titles_2 = [os.path.basename(p) for p in paths_2]
    print(f"Batch 2 contains: {titles_2}")
    show_images(images_2, titles_2, rows=1, cols=2)

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

```

Setting a random seed initializes the random number generator to a fixed starting state, ensuring that stochastic processes like data shuffling produce the exact same sequence of results every time the script is run. This repro-

ducibility is critical in deep learning because it allows researchers to isolate variables, confirming that changes in model performance are due to actual code improvements rather than random chance.

3.5 Skeleton Code

```
"""
ECE 60146 - Homework 2
Name: Triparna Mahata
Email: tmahata@purdue.edu

ImageNet Data Loading and Augmentation
"""
############################################################
import os
import torch
import numpy as np
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from PIL import Image
import time
import multiprocessing
multiprocessing.set_start_method('spawn', force=True)
############################################################
REQUIRED_CLASSES = {
    1: "goldfish",
    151: "Chihuahua",
    281: "tabby_cat",
    291: "lion",
    325: "sulphur_butterfly",
    386: "African_elephant",
    430: "basketball",
    466: "bullet_train",
    496: "Christmas_stocking",
    950: "orange"
}

def get_class_name(label):
    """Get class name from integer label."""
    return REQUIRED_CLASSES.get(label, f"class_{label}")
############################################################
# TRANSFORMS
transform_custom = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.2),
    transforms.RandomAffine(degrees=20, translate=(0.1,0.1), scale=(0.9,1.1)),
    transforms.Resize((224,224)),
    transforms.GaussianBlur(3,(0.3,1.5)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225])
])

transform_basic = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor()
])
############################################################
# OPTION A : Dataset class for provided subset
class ImageNetSubset(Dataset):
    """
    Dataset for loading the provided ImageNet subset.
    imagenet_subset/
                  1/    (goldfish)
                    00001.JPEG
                    ...
                151/    (Chihuahua)
```

```
                    00001.JPEG
                    ...
                ...
    Args:
        root(str): Path to imagenet_subset folder
        class_labels(list): List of integer labels to load (e.g. [1, 151, 281])
        images_per_class(int): Number of images to load per class
        transform (callable): Transform to apply to images
    """
    def __init__(self, root, class_labels, images_per_class = 5, transform = None):
        self.root = root
        self.class_labels = class_labels
        self.images_per_class = images_per_class
        self.transform = transform

        self.samples = []   # List of (image_path, label)
        self._load_samples()

        print (f"Loaded {len(self.samples)} images from {len(class_labels)} classes")

    def _load_samples(self):
        """Load image paths for each requested class."""
        # For each label in self.class_labels:
        #   1. Build path to class folder: os.path.join(self.root, str(label))
        #   2. List all .JPEG /.jpg /.png files in that folder
        #   3. Take first self.images_per_class images
        #   4. Append (image_path, label) to self.samples

        for label in self.class_labels:
            class_dir = os.path.join(self.root, str(label))
            all_files = sorted(os.listdir(class_dir))
            images = [f for f in all_files if f.lower().endswith(('.jpeg', '.jpg', '.png'))]
            selected_images = images[:self.images_per_class]
            for img_name in selected_images:
                img_path = os.path.join(class_dir, img_name)
                self.samples.append((img_path, label))

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

    def __getitem__(self, index):
        """
        Returns:
            image(Tensor): Transformed image
            label(int): Class label (e.g., 1 for goldfish)
        """
        # 1. Get image_path, label from self.samples[index]
        # 2. Load image using PIL: Image.open(path).convert('RGB')
        # 3. Apply self.transform if not None
        # 4. Return (image, label)

        image_path, label = self.samples[index]
        image = Image.open(image_path).convert('RGB')
        if self.transform:
            image = self.transform(image)
        return image, label
############################################################
# Custom dataset for my own images
class CustomDataset(Dataset):
    """Dataset for loading custom images from a folder."""
    def __init__(self, root, transform = None):
        self.root = root
        self.transform = transform

        # Load all image paths from root folder
        # Filter for: .jpg, .jpeg, .png, .bmp, .gif
```

```
        self.image_paths = []

        valid_extns = ('.jpg', '.jpeg', '.png', '.bmp', '.gif')

        for filename in sorted(os.listdir(self.root)):
            if filename.lower().endswith(valid_extns):
                full_path = os.path.join(self.root, filename)
                self.image_paths.append(full_path)

        print(f"Found {len(self.image_paths)} images in {root}")

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

    def __getitem__(self, index):
        img_path = self.image_paths[index]
        image = Image.open(img_path).convert('RGB')
        if self.transform:
            image = self.transform(image)
        return image, img_path
############################################################
# Utility Class
class RepeatedDataset(Dataset):
    def __init__(self, dataset, target_length=1000):
        self.dataset = dataset
        self.target_length = target_length
        self.original_length = len(dataset)

    def __len__(self):
        return self.target_length

    def __getitem__(self, index):
        return self.dataset[index % self.original_length]
############################################################
# Utility Functions
def denormalize(tensor, mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225]):
    """ Denormalize a tensor image for display."""
    mean = torch.tensor(mean).view(3,1,1)
    std = torch.tensor(std).view(3,1,1)
    return tensor * std + mean

def show_images(images, titles, rows, cols, figsize=(15,10), save_path=None):
    """Display a grid of images."""
    fig, axes = plt.subplots(rows, cols, figsize = figsize)
    axes = axes.flatten() if rows*cols > 1 else [axes]

    for i,(img,title) in enumerate(zip(images,titles)):
        if i >= len(axes):
            break
        if isinstance(img, torch.Tensor):
            if img.min() < 0 or img.max() > 1:
                img = denormalize(img)
            img = img.permute(1,2,0).numpy()

        img = np.clip(img,0,1)
        axes[i].imshow(img)
        axes[i].set_title(title,fontsize = 10)
        axes[i].axis('off')

    plt.tight_layout()
    if save_path:
        plt.savefig(save_path, dpi=150, bbox_inches='tight')
    plt.show()

def set_seed(seed = 60146):
    """Set random seed for reproducibility."""
```

```
    import random
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
############################################################
def main():
    # Required class labels
    required_labels = list(REQUIRED_CLASSES.keys())
    print("Required classes:", required_labels)
    for label in required_labels:
        print(f" {label}: {get_class_name(label)}")

    print("\n" + "=" * 60)

    ############ Task 1: Load ImageNet (50 images: 10 classes x 5 images) ############
    print("Task 1: Loading ImageNet")
    print("=" * 60)

    # Creating dataset using Option A
    imagenet_dataset = ImageNetSubset(root = './imagenet_subset',
                                      class_labels = required_labels,
                                      images_per_class = 5,
                                      transform = transform_custom)

    print("\n" + "=" * 60)

    ############ Task 2: Visualize ImageNet (1 image per class => 10 images) ############
    print("Task 2: Visualizing ImageNet")
    print("=" * 60)

    # Display 10 images (1 per class) in 2x5 grid
    # Save as 'imagenet_samples.png'
    images_display = []
    titles_display = []
    found_labels = set()

    for i in range(len(imagenet_dataset)):
        img, label = imagenet_dataset[i]

        if label not in found_labels:
            found_labels.add(label)
            images_display.append(img)
            titles_display.append(get_class_name(label))

        if len(found_labels) == len(required_labels):
            break

    show_images(
        images=images_display,
        titles=titles_display,
        rows=2,
        cols=5,
        save_path='imagenet_samples.png'
    )

    print("\n" + "=" * 60)

    ############ Task 3: Show augmentation effects ############
    print("Task 3: Augmentation comparison")
    print("=" * 60)

    # For 3 images, show original + 3 augmented versions
    idx_aug = np.random.choice(50, 3, replace=False)
    print(f"Showing augmentation for images at indices: {idx_aug}")
```

```
    aug_images = []
    aug_titles = []

    for idx in idx_aug:
        img_path, label = imagenet_dataset.samples[idx]
        class_name = get_class_name(label)

        pil_img = Image.open(img_path).convert('RGB')
        orig_tensor = transform_basic(pil_img)
        aug_images.append(orig_tensor)
        aug_titles.append(f"Original\n{class_name}")

        for k in range(3):
            aug_tensor = transform_custom(pil_img)
            aug_images.append(aug_tensor)
            aug_titles.append(f"Augmented Image {k+1}")

    show_images(
        images=aug_images,
        titles=aug_titles,
        rows=3,
        cols=4,
        figsize=(12, 10),
        save_path='augmentation_comparison.png'
    )

    print("\n" + "=" * 60)

    ############ Task 4: Custom dataset ############
    print("Task 4: Custom dataset")
    print("=" * 60)

    # Create CustomDataset, augment 20 images to 50
    # Display 10 samples, save as 'custom_samples.png'

    custom_root = './my_images'
    my_orig_dataset = CustomDataset(root=custom_root, transform=None)

    final_data = []

    for i in range(len(my_orig_dataset)):
        pil_img, _ = my_orig_dataset[i]
        tensor_img = transform_basic(pil_img)
        final_data.append((tensor_img, "Original"))

    aug_count = 0
    while len(final_data) < 50:
        idx = aug_count % len(my_orig_dataset)
        pil_img, _ = my_orig_dataset[idx]
        tensor_img = transform_custom(pil_img)
        final_data.append((tensor_img, "Augmented"))
        aug_count += 1

    sample_indices = np.random.choice(len(final_data), 10, replace=False)
    disp_imgs = [final_data[i][0] for i in sample_indices]
    disp_titles = [final_data[i][1] for i in sample_indices]

    show_images(images=disp_imgs,
                titles=disp_titles,
                rows=2,
                cols=5,
                save_path='custom_samples.png')

    print("\n" + "=" * 60)

    ############ Task 5: DataLoader Performance ############
```

```
print("Task 5: DataLoader Performance")
print("=" * 60)

# Compare loading times with different batch_size / num_workers
# Test: (16,0), (16,4), (64,0), (64,4)

test_dataset = RepeatedDataset(imagenet_dataset, target_length=1000)
print(f"Base images: {len(imagenet_dataset)}")
print(f"Performance dataset size: {len(test_dataset)}")
print("-" * 45)
print("Measuring __getitem__ loop time (Single Process)")
start_time = time.time()
for i in range(len(test_dataset)):
    _ = test_dataset[i]
end_time = time.time()
loop_duration = end_time - start_time
print(f"Time to load in simple loop: {loop_duration:.4f} sec")
print("-" * 45)

# Configurations: (batch_size, num_workers)
configs = [(16, 0),
           (16, 4),
           (64, 0),
           (64, 4)]

print(f"{'batch_size':<15} {'num_workers':<15} {'Time (sec)':<15}")
print("-" * 45)

for batch_size, num_workers in configs:
    loader = DataLoader(test_dataset,
                        batch_size = batch_size,
                        shuffle = True,
                        num_workers = num_workers)
    start_t = time.time()
    for batch in loader:
        pass
    end_t = time.time()
    duration = end_t - start_t
    print(f"{batch_size:<15} {num_workers:<15} {duration:.4f}")

print("\n" + "=" * 60)

############ Task 6: RGB Statistics ############
print("Task 6: RGB Statistics")
print("=" * 60)

# Compute min / max RGB before and after normalization
min_before = torch.tensor([float('inf'), float('inf'), float('inf')])
max_before = torch.tensor([float('-inf'), float('-inf'), float('-inf')])

min_after = torch.tensor([float('inf'), float('inf'), float('inf')])
max_after = torch.tensor([float('-inf'), float('-inf'), float('-inf')])

for i in range(len(imagenet_dataset)):
    img, _ = imagenet_dataset[i]
    flat_img = img.view(3, -1)

    current_min = flat_img.min(dim=1).values
    current_max = flat_img.max(dim=1).values

    min_after = torch.min(min_after, current_min)
    max_after = torch.max(max_after, current_max)

    img_unnorm = denormalize(img)
    flat_unnorm = img_unnorm.view(3, -1)

```

```
        current_min_un = flat_unnorm.min(dim=1).values
        current_max_un = flat_unnorm.max(dim=1).values

        min_before = torch.min(min_before, current_min_un)
        max_before = torch.max(max_before, current_max_un)

    channels = ['R', 'G', 'B']
    print("\nStatistics BEFORE Normalization:")
    print(f"{'Channel':<10} {'Min':<10} {'Max':<10}")
    print("-" * 35)
    for i, c in enumerate(channels):
        print(f"{c:<10} {min_before[i]:.4f}     {max_before[i]:.4f}")

    print("\nStatistics AFTER Normalization:")
    print(f"{'Channel':<10} {'Min':<10} {'Max':<10}")
    print("-" * 35)
    for i, c in enumerate(channels):
        print(f"{c:<10} {min_after[i]:.4f}     {max_after[i]:.4f}")

    print("\n" + "=" * 60)

    ############ Task 7: Reproducibility ############
    print("Task 7: Reproducibility")
    print("=" * 60)

    # Test with and without set_seed(60146)
    def get_first_batch_sample():
        loader = DataLoader(imagenet_dataset, batch_size=5, shuffle=True)
        images, labels = next(iter(loader))
        return images, labels.tolist()

    print("--- Experiment A: With set_seed(60146) ---")

    # Run 1
    set_seed(60146)
    imgs_1, labels_1 = get_first_batch_sample()

    # Run 2 (Resetting seed)
    set_seed(60146)
    imgs_2, labels_2 = get_first_batch_sample()

    # Compare Run 1 vs Run 2
    labels_match_12 = (labels_1 == labels_2)
    images_match_12 = torch.equal(imgs_1, imgs_2)

    if labels_match_12 and images_match_12:
        print("SUCCESS: Batch matches perfectly after resetting seed.")
    else:
        print("FAILURE: Mismatch detected despite fixed seed.")

    print("\n--- Experiment B: Without resetting seed ---")

    # Run 3 (No seed reset)
    imgs_3, labels_3 = get_first_batch_sample()

    # Compare Run 2 vs Run 3
    labels_match_23 = (labels_2 == labels_3)
    images_match_23 = torch.equal(imgs_2, imgs_3)

    if not labels_match_23 and not images_match_23:
        print("SUCCESS: Batch differs (as expected) when seed is not reset.")
    else:
        print(f"NOTE: Batch happened to match randomly.\n")

    print("\n" + "=" * 60)
    print ("Done!")
```

```
    print("\n" + "=" * 60)
##############################################################
if __name__ == "__main__":
    main()
```

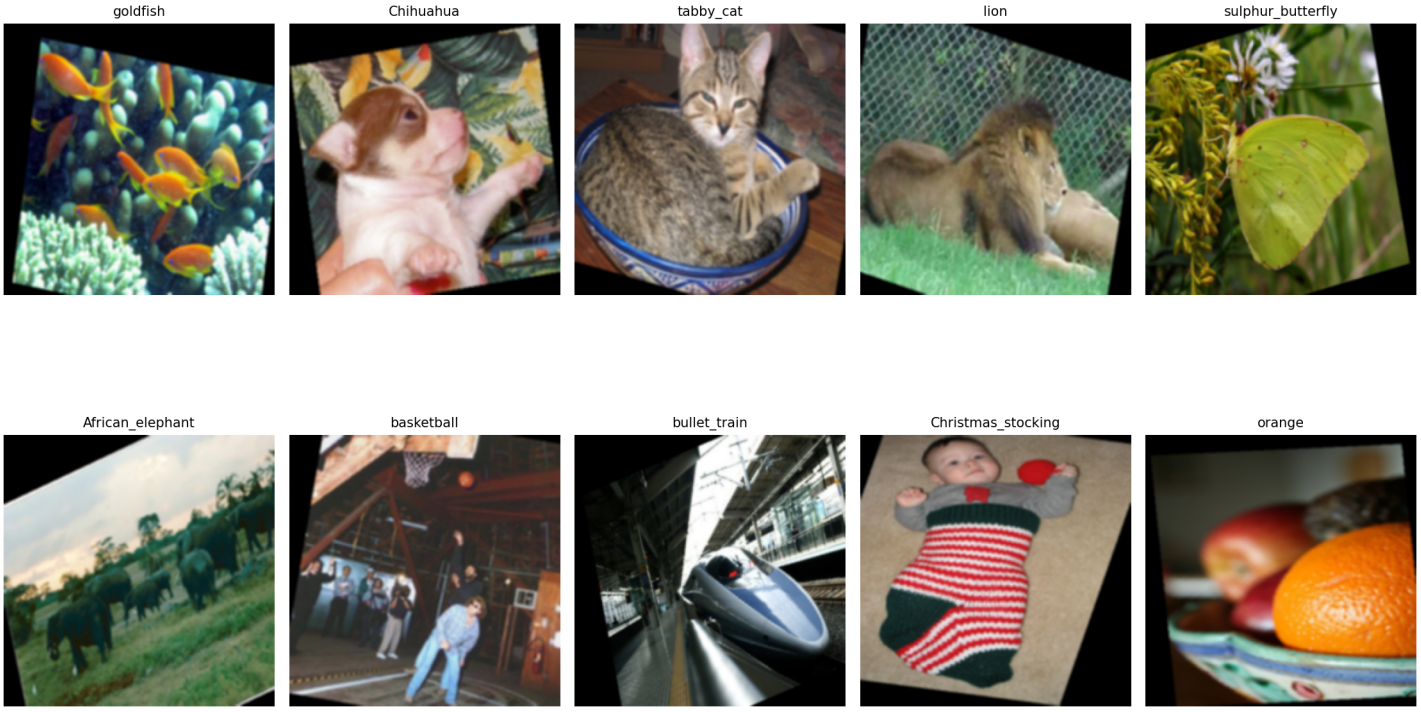
3.6 Task Summary

1. Load ImageNet: Implement the dataset class. Load 50 images (5 per class) from the 10 required classes.

```
Required classes: [1, 151, 281, 291, 325, 386, 430, 466, 496, 950]
 1: goldfish
 151: Chihuahua
 281: tabby_cat
 291: lion
 325: sulphur_butterfly
 386: African_elephant
 430: basketball
 466: bullet_train
 496: Christmas_stocking
 950: orange

============================================================
Task 1: Loading ImageNet
============================================================
Loaded 50 images from 10 classes

```

2. Visualize ImageNet: Display 10 images (one from each class) in a 2×5 grid.

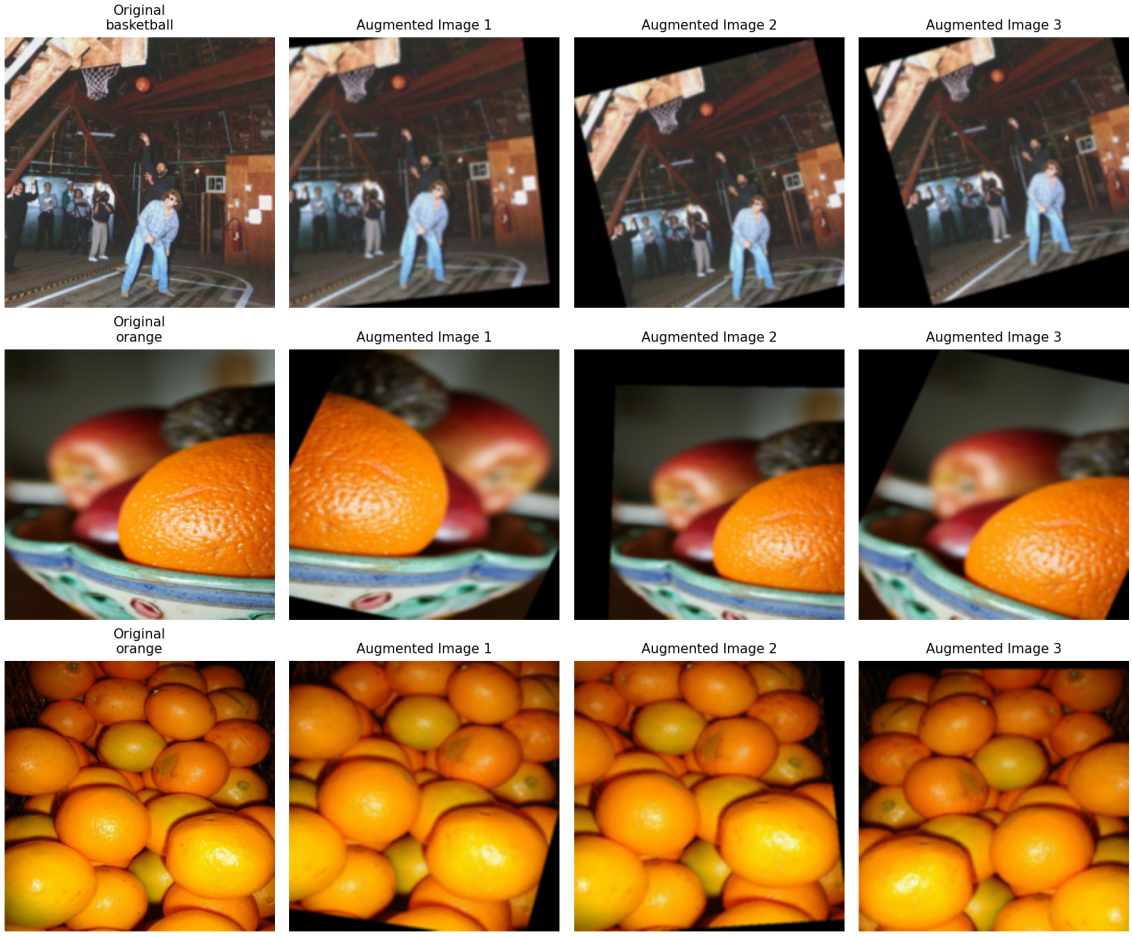
```
============================================================
Task 2: Visualizing ImageNet
============================================================

```



3. Augmentation Comparison: For 3 images, show original alongside 3 augmented versions.

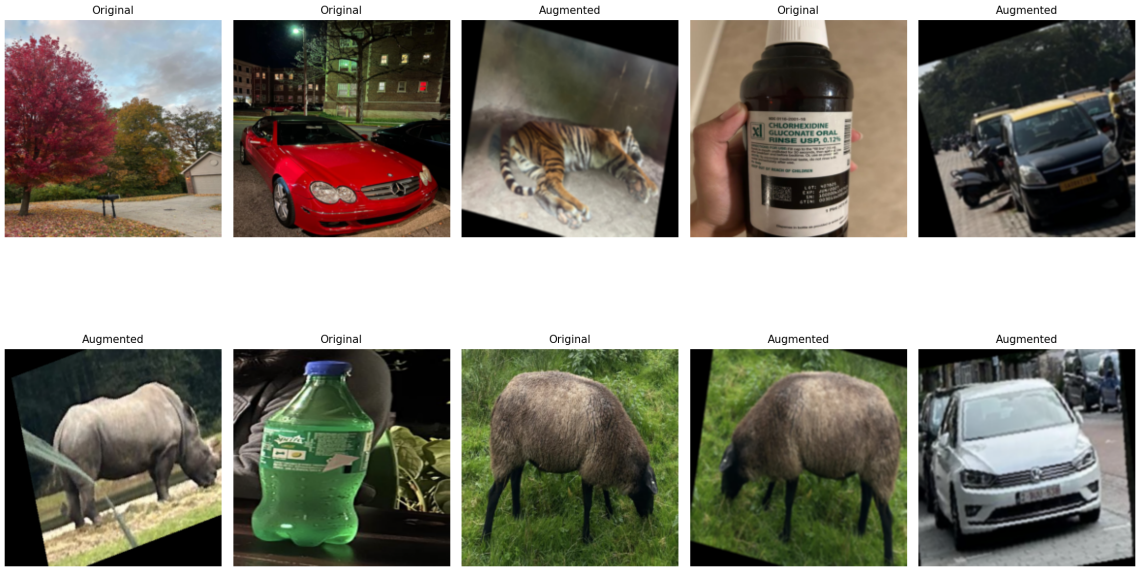
```
============================================================
Task 3: Augmentation comparison
============================================================
```

```
Showing augmentation for images at indices: [30 45 46]

```



4. Custom Dataset: Collect 20 images, augment to 50, display 10 samples.

```
============================================================
Task 4: Custom dataset
============================================================
Found 20 images in ./my_images

```



5. DataLoader Performance: Compare loading times.

```
============================================================
Task 5: DataLoader Performance
============================================================
Base images: 50
Performance dataset size: 1000
--------------------------------------------
Measuring __getitem__ loop time (Single Process)
Time to load in simple loop: 3.1491 sec
--------------------------------------------
batch_size      num_workers     Time (sec)
--------------------------------------------
16              0               3.1638
16              4               22.5881
64              0               2.9990
64              4               22.4135

```

6. RGB Statistics: Compute min/max per channel before/after normalization.

```
============================================================
Task 6: RGB Statistics
============================================================

Statistics BEFORE Normalization:
Channel    Min         Max
----------------------------------
R          0.0000      1.0000
G          0.0000      1.0000
B          0.0000      1.0000

Statistics AFTER Normalization:
Channel    Min         Max
----------------------------------
R          -2.1179      2.2489
G          -2.0357      2.4286
B          -1.8044      2.6400

```

7. Reproducibility: Demonstrate effect of random seed on batch ordering.

```
============================================================
Task 7: Reproducibility
============================================================
--- Experiment A: With set_seed(60146) ---
SUCCESS: Batch matches perfectly after resetting seed.

--- Experiment B: Without resetting seed ---
SUCCESS: Batch differs (as expected) when seed is not reset.

```