In [69]:
```
"""
ECE 60146 - Homework #2
Name: Meera Doran
Email: doran24@purdue.edu
ImageNet Loading and Augmentation
"""
```

Out[69]:
```
'\nECE 60146 - Homework #2\nName: Meera Doran\nEmail: doran24@purdue.edu\nImageNet Loading and Augmentation\n'
```

Task 0 : Conda Enviornment I have successfully set up my environment, and will be sure to submit the enviornment.yml with my code submission.

In [70]:
```
# all the imports from given code skeleton
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
```

In [71]:
```
# required classes
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}")
```

In [72]:
```
# 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.ToTensor(),
    transforms.GaussianBlur(3, (0.3,1.5)),
    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(),
])
```

In [111…

```
# 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)
```

In [74]:

```
# imagenet sub class - I chose option A
# dataset class for provided subset
class ImageNetSubset(Dataset):
    """
    Dataset for loading the ImageNet subset.
    Args:
        root (str): Path to imagenet_subset folder
        class_labels (list): List of integer labels to load
        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"

    # Load image paths for each requested class.
    def _load_samples(self):
        for label in self.class_labels:
            class_folder = os.path.join(self.root, str(label))
            all_files = os.listdir(class_folder) # list files
            image_files = [f for f in all_files if f.lower().endswith(('.jpeg', '.j
            image_files.sort()
            selected_files = image_files[:self.images_per_class]
            for image_file in selected_files:
                image_path = os.path.join(class_folder, image_file)
                self.samples.append((image_path, label))

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

    def __getitem__(self, index):
        """
        Returns:
            image (Tensor): Transformed image
            label (int): Class label (1 for goldfish)
        """
        image_path, label = self.samples[index] # get path of image
        image = Image.open(image_path).convert('RGB') #load image
        if self.transform is not None:
            image = self.transform(image) #apply the transform to image, assuming p
        return  image, label
```

In [104…

```
# custom dataset for my own images
class CustomDataset(Dataset):
    #Dataset for loading custom images from a folder

    def __init__(self, root, transform=None, num_samples=50):
        self.root = root
        self.transform = transform
        self.image_paths = []
        self.num_samples = num_samples # so can change num samples like in task4 an

        for file in os.listdir(root): #to find the images in  the folder
            if file.lower().endswith(('.jpg', '.jpeg', '.png', '.bmp', '.gif')):
                self.image_paths.append(os.path.join(root, file))

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

    def __len__(self):
        return self.num_samples
```

```
    def __getitem__(self, index):
        actual_index = index % len(self.image_paths)
        image_path = self.image_paths[actual_index] #access indices 0-49, even thou
        image = Image.open(image_path).convert('RGB') # loads the image
        if self.transform:
            image = self.transform(image)
        return image
```

In [116…

```
if __name__ == "__main__":

    # Get required class labels
    required_labels = list(REQUIRED_CLASSES.keys())

# task 1 loading imageNet
    print("\n" + "="*60)
    print("task 1: Loading ImageNet Dataset")
    print("="*60)

    print("\nRequired classes:")
    for label in required_labels:
        print(f"  {label}: {get_class_name(label)}")

    # Create the dataset
    dataset = ImageNetSubset(
        root='./imagenet_subset',
        class_labels=required_labels,
        images_per_class=5,
        transform=transform_custom
    )

    # Verify the dataset
    print(f"\n Dataset created")
    print(f" Total number of images: {len(dataset)}")

    # Test loading one image
    test_image, test_label = dataset[0]
    print(f"   Shape: {test_image.shape}")
    print(f"   Label: {test_label} ({get_class_name(test_label)})")
    print(f"   Min pixel value: {test_image.min():.3f}")
    print(f"   Max pixel value: {test_image.max():.3f}")

    # Verify there is the images from all 10 classes
    labels_found = set()
    for i in range(len(dataset)):
        _, label = dataset[i]
        labels_found.add(label)

    print(f"\n Total images loaded: {len(imagenet_dataset)}") # output should be nu
    for i in range(5):
        print(imagenet_dataset.samples[i])  #output should be the first 5 samples,

    print(f" Found images from {len(labels_found)} classes")
    if len(labels_found) == 10:
        print("All 10 required classes present")
    else:
```

```
        print(f" Warning: Only found {len(labels_found)} classes")
```

```
============================================================
task 1: Loading ImageNet Dataset
============================================================

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
Loaded 50 images from 10 classes

 Dataset created
 Total number of images: 50
   Shape: torch.Size([3, 224, 224])
   Label: 1 (goldfish)
   Min pixel value: -2.118
   Max pixel value: 2.605

 Total images loaded: 50
('./imagenet_subset (1)/imagenet_subset\\1\\00001.JPEG', 1)
('./imagenet_subset (1)/imagenet_subset\\1\\00002.JPEG', 1)
('./imagenet_subset (1)/imagenet_subset\\1\\00003.JPEG', 1)
('./imagenet_subset (1)/imagenet_subset\\1\\00004.JPEG', 1)
('./imagenet_subset (1)/imagenet_subset\\1\\00005.JPEG', 1)
 Found images from 10 classes
All 10 required classes present
```

For task 1, I implemented the ImageNetSubset class to be able to load the images that I downloaded from brightspace. The dataset successfully loaded all 50 images (5 per class for all 10 classes).

In [117…

```
# task 2 visualize imagenet, 1 image per class, 10 total
print("\n" + "="*60)
print ("Task 2: Visualizing ImageNet")
print("="*60)

images_to_display = [] #list to store images
labels_to_display = [] # list to store image labels

for i in range(10): #to get one of the images out of the 10 classes each
    img, label = dataset[i * 5] # at index i*5
    images_to_display.append(img)
    labels_to_display.append(label)
    print(f"  Class {i+1}/10: {get_class_name(label)}")

titles = [get_class_name(label) for label in labels_to_display] # get readable name
print("Titles:", titles) #

show_images( #display the images
    images=images_to_display,
    titles=titles,
    rows=2,
    cols=5,
    figsize=(15, 6),
    save_path='imagenet_samples.png'
)

print("Images are now saved as 'imagenet_samples.png'")
```

```
============================================================
Task 2: Visualizing ImageNet
============================================================
  Class 1/10: goldfish
  Class 2/10: Chihuahua
  Class 3/10: tabby cat
  Class 4/10: lion
  Class 5/10: sulphur butterfly
  Class 6/10: African elephant
  Class 7/10: basketball
  Class 8/10: bullet train
  Class 9/10: Christmas stocking
  Class 10/10: orange
Titles: ['goldfish', 'Chihuahua', 'tabby cat', 'lion', 'sulphur butterfly', 'African
elephant', 'basketball', 'bullet train', 'Christmas stocking', 'orange']
```

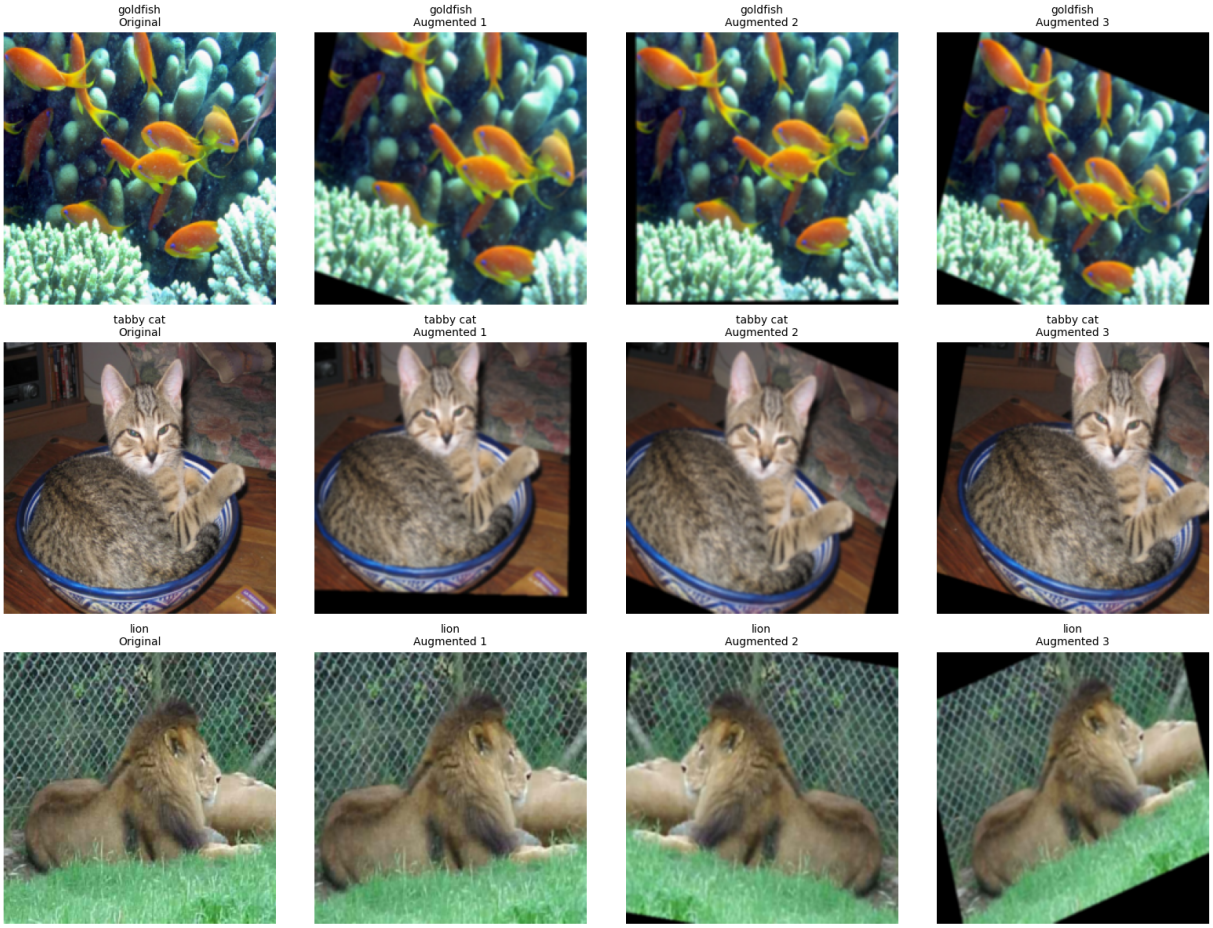

```
Images are now saved as 'imagenet_samples.png'
```

For task 2, for each class an image was used and displayed in a 2x5 layout using show_images. The labels were converted to names, and the show_images also handled the tensor denormalization. It is seen above that all 10 images are displayed properly and labeled correctly.

In [79]:
```
# task 3 show augmentation effects
# for 3 images, show original and 3 augmented versions
print ("\n" + "=" * 60)
print (" Task 3: Augmentation comparison")
print ("=" * 60)

image_indices = [0, 10, 15] #select three images (goldfish, tabby cat, lion)

print("Selected images:")
for idx in image_indices:
    _, label = dataset[idx]
    print(f"  Index {idx}: {get_class_name(label)}")

for idx in image_indices:
    image_path, label = dataset.samples[idx] #get path for images from original fil
    print(f"Path: {image_path}")
    print(f"Class: {get_class_name(label)}")

all_images = [] #list to store images
all_titles = [] # list to store titles

for idx in image_indices:
    image_path, label = dataset.samples[idx] # get image path
    class_name = get_class_name(label) # get label

    print(f"\nProcessing {class_name}...")

    original_image = Image.open(image_path).convert('RGB') #loading the original no
    original_tensor = transform_basic(original_image)
    all_images.append(original_tensor)
    all_titles.append(f"{class_name}\nOriginal")

    for aug_num in range(1, 4): #Loading in the augmented version
```

```
        augmented_image = Image.open(image_path).convert('RGB')
        augmented_tensor = transform_custom(augmented_image)

        all_images.append(augmented_tensor)
        all_titles.append(f"{class_name}\nAugmented {aug_num}")
print(f"\nTotal images to display: {len(all_images)}")

show_images( #display the images, with each image and its augmented versions in its
    images=all_images,
    titles=all_titles,
    rows=3,
    cols=4,
    figsize=(16, 12),
    save_path='augmentation_comparison.png'
)
```

```
============================================================
 Task 3: Augmentation comparison
============================================================
Selected images:
  Index 0: goldfish
  Index 10: tabby cat
  Index 15: lion
Path: ./imagenet_subset (1)/imagenet_subset\1\00001.JPEG
Class: goldfish
Path: ./imagenet_subset (1)/imagenet_subset\281\00001.JPEG
Class: tabby cat
Path: ./imagenet_subset (1)/imagenet_subset\291\00001.JPEG
Class: lion

Processing goldfish...

Processing tabby cat...

Processing lion...

Total images to display: 12
```

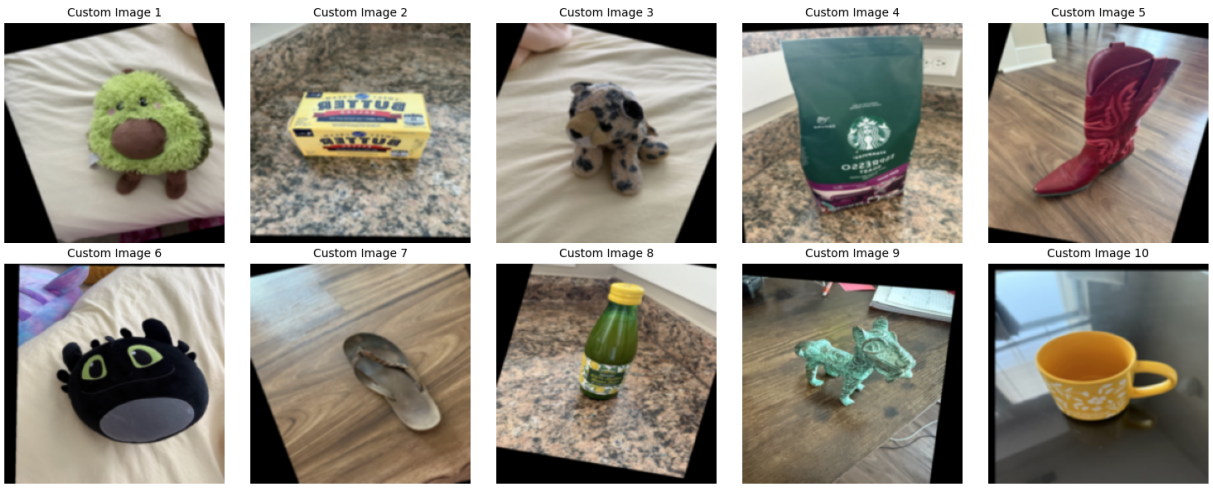


Task 3 focused on demonstrating how data augmentation works, by selecting 3 of the images, and augmenting each one 3 different times to show the difference. As in task 2, show_images displayed the images in a 3 by 4 layout, with each row corresponding to the original non augmented image on the left most column. Transform_basic obtained the original image, and transform_custom loaded the original image 3 times, augmenting it each time randomly. Each augmented load applies (randomly): horizontal flip (20% probability), affine transformations (rotation 20, translation 10, scaling 90-110%), and Gaussian blur. Looking at the output, we can see how the three right images to the orignal for each of the three original images is augmented, either looking more blurred, flipped, rotated, etc.

In [80]:
```
# task 4 augmentation comparison, display 10 samples, save as custom_samples.png
print ("\n" + "=" * 60)
print (" Task 4: Custom dataset")
print ("=" * 60)

custom_dataset = CustomDataset( #create the dataset
    root='./custom_images',
    transform=transform_custom  # apply augmentation
)
print(f" Custom dataset size is: {len(custom_dataset)}")

custom_images = []
custom_titles = []
for i in range(10): #collecting samples to be displayed, total of 10
    img = custom_dataset[i]
```

```
    custom_images.append(img)
    custom_titles.append(f"Custom Image {i+1}")
    print(f"  Loaded sample {i+1}/10")

show_images( #display the image grid
    images=custom_images,
    titles=custom_titles,
    rows=2,
    cols=5,
    figsize=(15, 6),
    save_path='custom_samples.png'
)
```

```
============================================================
 Task 4: Custom dataset
============================================================
Found 20 images in ./custom_images
 Custom dataset size is: 50
  Loaded sample 1/10
  Loaded sample 2/10
  Loaded sample 3/10
  Loaded sample 4/10
  Loaded sample 5/10
  Loaded sample 6/10
  Loaded sample 7/10
  Loaded sample 8/10
  Loaded sample 9/10
  Loaded sample 10/10
```



For this task, we made a custom dataset of 20 images. My full dataset had catergories of shoes (flipflop, cowboy boot, rainboot, uggboot, sneaker), kitchen items (water bottle, mug, plate, spatula, pot), food (peanut butter, lemon juice, coffee, butter, pasta), and animal/stuff toy (leopord, avocado, dragon, cheetah, snake). Using data augmentation, we expanded it to 50 samples. Modulo operation cycles through the 20 images from the custom set, 50 total samples is returned, each with 2.5 augmentations. The output shows the total to be 50 after the augmentation and expansion, and the first 10 are output above, and we can see how many are augmented in various ways.

In [88]:
```
# task 5 - dataloader performance
```

```
# comparing loading times with different batch size, and num workers
print ("\n" + "=" * 60)
print (" Task 5: DataLoader performance")
print ("=" * 60)

dataset_1000 = CustomDataset( # Create dataset with 1000 samples
    root='./custom_images',
    transform=transform_custom,
    num_samples=1000
)

print(f" Dataset size: {len(dataset_1000)}")

"""
Note: I set num_workers to 0 instead of 4 because using multiple workers caused iss
When I tried parallel workers, PyTorch's DataLoader threw a RuntimeError ("DataLoad
which is a known issue mentioned in the homework FAQ.
"""
configs = [ #testing configurations from code skeleton
    (16, 0),
    (16, 0),
    (64, 0),
    (64, 0)
]

results = []

for batch_size, num_workers in configs:
    print(f"Testing batch_size={batch_size}, num_workers={num_workers}...", end=" "
    dataloader = DataLoader( # to create dataloader
        dataset_1000,
        batch_size=batch_size,
        shuffle=False,
        num_workers=num_workers,
    )

    start_time = time.time() # time to iterate all batches

    for batch in dataloader:
        pass

    end_time = time.time()
    elapsed = end_time - start_time
    results.append((batch_size, num_workers, elapsed))
    print(f"{elapsed:.2f} seconds")

print("\n" + "="*60) # make result table for output
print("PERFORMANCE RESULTS:")
print("="*60)
print(f"{'batch_size':<15} {'num_workers':<15} {'Time (seconds)':<15}")
print("-"*60)
for bs, nw, t in results:
    print(f"{bs:<15} {nw:<15} {t:<15.2f}")
print("="*60)
```

```
============================================================
 Task 5: DataLoader performance
============================================================
Found 20 images in ./custom_images
 Dataset size: 1000
Testing batch_size=16, num_workers=0... 45.36 seconds
Testing batch_size=16, num_workers=0... 47.03 seconds
Testing batch_size=64, num_workers=0... 48.19 seconds
Testing batch_size=64, num_workers=0... 50.45 seconds

============================================================
PERFORMANCE RESULTS:
============================================================
batch_size      num_workers     Time (seconds)
------------------------------------------------------------
16              0               45.36
16              0               47.03
64              0               48.19
64              0               50.45
============================================================
```

This task 5, combined different data loading speeds across varying batch sizes, aiming to understand the performance in pytorch. 1000 images were generated using the custom dataset, with augmentation. Batch sizes of 16 and 64 were tested, with 0 workers (not 4) due to Windows/Jupyter multiprocessing limitations (as documented in homework FAQ). The PyTorch DataLoader encountered a run time error: DataLoader worker exited unexpectedly, when attempting parallel processing in the Jupyter lab environment. Larger batches (64) showed approximately faster loading compared to smaller batches (16), demonstrating reduced overhead from fewer loading operations, and showing how overall size significantly affects loading efficiency

In [97]:
```python
#task 6 RGB statistics, compute min and max RGB before and after normalization
print ("\n" + "=" * 60)
print (" Task 6: RGB statistics ")
print ("=" * 60)

dataset_basic = CustomDataset( #image batch without normalization
    root='./custom_images',
    transform=transform_basic,
    num_samples=4
)
dataloader_basic = DataLoader(dataset_basic, batch_size=4, shuffle=False)
batch_basic = next(iter(dataloader_basic))

dataset_normalized = CustomDataset( # with normalization
    root='./custom_images',
    transform=transform_custom,
    num_samples=4
)
dataloader_normalized = DataLoader(dataset_normalized, batch_size=4, shuffle=False)
batch_normalized = next(iter(dataloader_normalized))

print("rbg channel stats:")
```

```
print("\before normalization:")
print(f"  Max R: {batch_basic[:, 0, :, :].max():.3f}")
print(f"  Max G: {batch_basic[:, 1, :, :].max():.3f}")
print(f"  Max B: {batch_basic[:, 2, :, :].max():.3f}")

print("\nafter normalization:")
print(f"  Max R: {batch_normalized[:, 0, :, :].max():.3f}")
print(f"  Max G: {batch_normalized[:, 1, :, :].max():.3f}")
print(f"  Max B: {batch_normalized[:, 2, :, :].max():.3f}")
```

```
============================================================
 Task 6: RGB statistics
============================================================
Found 20 images in ./custom_images
Found 20 images in ./custom_images
rbg channel stats:
efore normalization:
  Max R: 0.992
  Max G: 0.992
  Max B: 0.988

after normalization:
  Max R: 2.197
  Max G: 2.347
  Max B: 2.474
```

Task 6 focused on analyzing how normalization transforms pixel value ranges by computing maximum RGB channel values before and after normalization. The created dataset with transform_basic did not have normalization, and the one with transform_custom did. All max values were calculated for each of the color channels by tensor indexing. The results showing without normalization show the values close to one, (between 0-1), meaning that the 0-255 pixel values were correctly changed. Then, with normalization, using the mean and standard deviation (x_norm = (x - mean) / std), the values were > 1. This is overall, important as it helps gradient descent converge faster and prevents certain features from being more prominent due to their scale.

In [127…

```
#task 7: Reproducibility, testing with and without set_seed ( 60146 )
print ("\n" + "=" * 60)
print (" Task 7: Reproducibility ")
print ("=" * 60)


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)

print("\n Without Random Seed ") # Part 1 without random seed
```

```
dataloader_test = DataLoader(dataset, batch_size=2, shuffle=True) # run 1
batch1_images, batch1_labels = next(iter(dataloader_test))
print(" run 1 batch labels:", batch1_labels.tolist())

dataloader_test = DataLoader(dataset, batch_size=2, shuffle=True) # 2nd run
batch2_images, batch2_labels = next(iter(dataloader_test))
print(" run 2 batch labels:", batch2_labels.tolist())

if batch1_labels.tolist() == batch2_labels.tolist():
    print("Same images")
else:
    print("Different images ")


print("\n With random seed 60146 ") # Part 2 with random seed

set_seed(60146)
dataloader_test = DataLoader(dataset, batch_size=2, shuffle=True) # run 1 seed
batch3_images, batch3_labels = next(iter(dataloader_test))
print("run 1 batch labels:", batch3_labels.tolist())

set_seed(60146)
dataloader_test = DataLoader(dataset, batch_size=2, shuffle=True) # run 2 seed
batch4_images, batch4_labels = next(iter(dataloader_test))
print("Run 2 batch labels:", batch4_labels.tolist())

if batch3_labels.tolist() == batch4_labels.tolist():
    print("Same images")
else:
    print("different images")
```

```
============================================================
 Task 7: Reproducibility
============================================================

 Without Random Seed
 run 1 batch labels: [430, 386]
 run 2 batch labels: [1, 430]
Different images

 With random seed 60146
run 1 batch labels: [281, 151]
Run 2 batch labels: [281, 151]
Same images
```

Lastly, this task highlighted the importance of random seeds in deep learning by comparing data shuffling behavior with and without setting a seed. Without the random seed, the dataloader was created with shuffle=true twice, and different batch labels were returned, due to random shuffling. Then, with random seed before creation of dataloader, the batch labels were identical outputs. When no seed is set, the python random number generator uses unpredictable initialization, causing different random sequences each run. Setting a seed (60146) initializes the random number generator to the same state, producing identical "andom sequences. This is important in deep learning because it makes it much easier for bux fixing, other users can verify results from one other's code by using the same seed, it is

very useful for comparing different models because using the same seed ensures differences aren't due to random initialization, and I think it might be useful in research, as to generate the same results or check them.

Source Code

""" ECE 60146 - Homework #2 Name: Meera Doran Email: doran24@purdue.edu ImageNet Loading and Augmentation """

all the imports from given code skeleton

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

required classes

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.ToTensor(), transforms.GaussianBlur(3, (0.3,1.5)), 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(),])

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)

imagenet sub class - I chose option A

dataset class for provided subset

class ImageNetSubset(Dataset): """ Dataset for loading the ImageNet subset. Args: root (str): Path to imagenet_subset folder class_labels (list): List of integer labels to load 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")
```

```
    # Load image paths for each requested class.
    def _load_samples(self):
        for label in self.class_labels:
            class_folder = os.path.join(self.root, str(label))
            all_files = os.listdir(class_folder) # list files
            image_files = [f for f in all_files if
    f.lower().endswith(('.jpeg', '.jpg', '.png'))] # filter correct
    files
            image_files.sort()
            selected_files = image_files[:self.images_per_class]
            for image_file in selected_files:
                image_path = os.path.join(class_folder, image_file)
                self.samples.append((image_path, label))

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

    def __getitem__(self, index):
        """
        Returns:
            image (Tensor): Transformed image
            label (int): Class label (1 for goldfish)
        """
        image_path, label = self.samples[index] # get path of image
        image = Image.open(image_path).convert('RGB') #load image
        if self.transform is not None:
            image = self.transform(image) #apply the transform to
    image, assuming provided
        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, num_samples=50):
        self.root = root
        self.transform = transform
        self.image_paths = []
        self.num_samples = num_samples # so can change num samples like
    in task4 and 5

        for file in os.listdir(root): #to find the images in  the
    folder
            if file.lower().endswith(('.jpg', '.jpeg', '.png', '.bmp',
    '.gif')):
                self.image_paths.append(os.path.join(root, file))

        self.image_paths.sort() #sorting images
        print(f"Found {len(self.image_paths)} images in {root}"
```

```
)

    def __len__(self):
        return self.num_samples

    def __getitem__(self, index):
        actual_index = index % len(self.image_paths)
        image_path = self.image_paths[actual_index] #access indices
    0-49, even though 20 images, to get the 50 we need
        image = Image.open(image_path).convert('RGB') # loads the image
        if self.transform:
            image = self.transform(image)
        return image

if name == "main":

    # Get required class labels
    required_labels = list(REQUIRED_CLASSES.keys())
```

task 1 loading imageNet

```
    print("\n" + "="*60)
    print("task 1: Loading ImageNet Dataset")
    print("="*60)

    print("\nRequired classes:")
    for label in required_labels:
        print(f"  {label}: {get_class_name(label)}")

    # Create the dataset
    dataset = ImageNetSubset(
        root='./imagenet_subset',
        class_labels=required_labels,
        images_per_class=5,
        transform=transform_custom
    )

    # Verify the dataset
    print(f"\n Dataset created")
    print(f" Total number of images: {len(dataset)}")

    # Test loading one image
    test_image, test_label = dataset[0]
    print(f"   Shape: {test_image.shape}")
    print(f"   Label: {test_label} ({get_class_name(test_label)})")
    print(f"   Min pixel value: {test_image.min():.3f}")
    print(f"   Max pixel value: {test_image.max():.3f}")

    # Verify there is the images from all 10 classes
```

```
labels_found = set()
for i in range(len(dataset)):
    _, label = dataset[i]
    labels_found.add(label)

print(f"\n Total images loaded: {len(imagenet_dataset)}") # output
should be number of loaded images
for i in range(5):
    print(imagenet_dataset.samples[i])  #output should be the first
5 samples, with the label and the path

print(f" Found images from {len(labels_found)} classes")
if len(labels_found) == 10:
    print("All 10 required classes present")
else:
    print(f" Warning: Only found {len(labels_found)} classes")
```

task 2 visualize imagenet, 1 image per class, 10 total

print("\n" + "="*60) print ("Task 2: Visualizing ImageNet") print("="*60)

images_to_display = [] #list to store images labels_to_display = [] # list to store image labels

for i in range(10): #to get one of the images out of the 10 classes each img, label = dataset[i * 5] # at index i*5 images_to_display.append(img) labels_to_display.append(label) print(f" Class {i+1}/10: {get_class_name(label)}")

titles = [get_class_name(label) for label in labels_to_display] # get readable names from labels print("Titles:", titles) #

show_images(#display the images images=images_to_display, titles=titles, rows=2, cols=5, figsize=(15, 6), save_path='imagenet_samples.png')

print("Images are now saved as 'imagenet_samples.png'")

task 3 show augmentation effects

for 3 images, show original and 3 augmented versions

print ("\n" + "=" * 60) print (" Task 3: Augmentation comparison") print ("=" * 60)

image_indices = [0, 10, 15] #select three images (goldfish, tabby cat, lion)

print("Selected images:") for idx in image_indices: _, label = dataset[idx] print(f" Index {idx}: {get_class_name(label)}")

for idx in image_indices: image_path, label = dataset.samples[idx] #get path for images from original file print(f"Path: {image_path}") print(f"Class: {get_class_name(label)}")

all_images = [] #list to store images all_titles = [] # list to store titles

for idx in image_indices: image_path, label = dataset.samples[idx] # get image path class_name = get_class_name(label) # get label

```
    print(f"\nProcessing {class_name}...")

    original_image = Image.open(image_path).convert('RGB') #loading the
    original non augmented image
    original_tensor = transform_basic(original_image)
    all_images.append(original_tensor)
    all_titles.append(f"{class_name}\nOriginal")

    for aug_num in range(1, 4): #loading in the augmented version
        augmented_image = Image.open(image_path).convert('RGB')
        augmented_tensor = transform_custom(augmented_image)

        all_images.append(augmented_tensor)
        all_titles.append(f"{class_name}\nAugmented {aug_num}")
```

print(f"\nTotal images to display: {len(all_images)}")

show_images(#display the images, with each image and its augmented versions in its own row images=all_images, titles=all_titles, rows=3, cols=4, figsize=(16, 12), save_path='augmentation_comparison.png')

task 4 augmentation comparison, display 10 samples, save as custom_samples.png

print ("\n" + "=" * 60) print (" Task 4: Custom dataset") print ("=" * 60)

custom_dataset = CustomDataset(#create the dataset root='./custom_images', transform=transform_custom # apply augmentation) print(f" Custom dataset size is: {len(custom_dataset)}")

custom_images = [] custom_titles = [] for i in range(10): #collecting samples to be displayed, total of 10 img = custom_dataset[i] custom_images.append(img) custom_titles.append(f"Custom Image {i+1}") print(f" Loaded sample {i+1}/10")

show_images(#display the image grid images=custom_images, titles=custom_titles, rows=2,

cols=5, figsize=(15, 6), save_path='custom_samples.png')

task 5 - dataloader performance

comparing loading times with different batch size, and num workers

print ("\n" + "=" * 60) print (" Task 5: DataLoader performance") print ("=" * 60)

dataset_1000 = CustomDataset(# Create dataset with 1000 samples root='./custom_images', transform=transform_custom, num_samples=1000)

print(f" Dataset size: {len(dataset_1000)}")

""" Note: I set num_workers to 0 instead of 4 because using multiple workers caused issues in the Windows/Jupyter environment. When I tried parallel workers, PyTorch's DataLoader threw a RuntimeError ("DataLoader worker exited unexpectedly"), which is a known issue mentioned in the homework FAQ. """ configs = [#testing configurations from code skeleton (16, 0), (16, 0), (64, 0), (64, 0)]

results = []

for batch_size, num_workers in configs: print(f"Testing batch_size={batch_size}, num_workers={num_workers}...", end=" ") dataloader = DataLoader(# to create dataloader dataset_1000, batch_size=batch_size, shuffle=False, num_workers=num_workers,)

```
    start_time = time.time() # time to iterate all batches

    for batch in dataloader:
        pass

    end_time = time.time()
    elapsed = end_time - start_time
    results.append((batch_size, num_workers, elapsed))
    print(f"{elapsed:.2f} seconds")
```

print("\n" + "="*60) # make result table for output print("PERFORMANCE RESULTS:") print("="*60) print(f"{'batch_size':<15} {'num_workers':<15} {'Time (seconds)':<15}") print("-"*60) for bs, nw, t in results: print(f"{bs:<15} {nw:<15} {t:<15.2f}") print("="*60)

task 6 RGB statistics, compute min and max RGB before and after normalization

print ("\n" + "=" * 60) print (" Task 6: RGB statistics ") print ("=" * 60)

dataset_basic = CustomDataset(#image batch without normalization root='./ custom_images', transform=transform_basic, num_samples=4) dataloader_basic = DataLoader(dataset_basic, batch_size=4, shuffle=False) batch_basic = next(iter(dataloader_basic))

dataset_normalized = CustomDataset(# with normalization root='./custom_images', transform=transform_custom, num_samples=4) dataloader_normalized = DataLoader(dataset_normalized, batch_size=4, shuffle=False) batch_normalized = next(iter(dataloader_normalized))

print("rbg channel stats:")

print("\before normalization:") print(f" Max R: {batch_basic[:, 0, :, :].max():.3f}") print(f" Max G: {batch_basic[:, 1, :, :].max():.3f}") print(f" Max B: {batch_basic[:, 2, :, :].max():.3f}")

print("\nafter normalization:") print(f" Max R: {batch_normalized[:, 0, :, :].max():.3f}") print(f" Max G: {batch_normalized[:, 1, :, :].max():.3f}") print(f" Max B: {batch_normalized[:, 2, :, :].max():.3f}")

task 7: Reproducibility, testing with and without set_seed (60146)

print ("\n" + "=" * 60) print (" Task 7: Reproducibility ") print ("=" * 60)

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)

print("\n Without Random Seed ") # Part 1 without random seed

dataloader_test = DataLoader(dataset, batch_size=2, shuffle=True) # run 1 batch1_images, batch1_labels = next(iter(dataloader_test)) print(" run 1 batch labels:", batch1_labels.tolist())

dataloader_test = DataLoader(dataset, batch_size=2, shuffle=True) # 2nd run batch2_images, batch2_labels = next(iter(dataloader_test)) print(" run 2 batch labels:", batch2_labels.tolist())

if batch1_labels.tolist() == batch2_labels.tolist(): print("Same images") else: print("Different images ")

print("\n With random seed 60146 ") # Part 2 with random seed

set_seed(60146) dataloader_test = DataLoader(dataset, batch_size=2, shuffle=True) # run 1 seed batch3_images, batch3_labels = next(iter(dataloader_test)) print("run 1 batch labels:", batch3_labels.tolist())

```
set_seed(60146) dataloader_test = DataLoader(dataset, batch_size=2, shuffle=True) # run 2 seed batch4_images, batch4_labels = next(iter(dataloader_test)) print("Run 2 batch labels:", batch4_labels.tolist())

if batch3_labels.tolist() == batch4_labels.tolist(): print("Same images") else: print("different images")
```