ARI2129 - Group Project 2024

Part B - Technical Paper

Name: **Andrea F. Lucas, Thomas Koppens & Sean D. Muscat**
Course: ARI2129 - Artificial Intelligence (AI)
ID No: 0279704L, 029204L & 0172004L

Distribution of Work - Technical Paper

Section	**Responsible Person**
Project Overview & README.md	Andrea F. Lucas
TensorFlow (Q3-a)	Andrea F. Lucas
PyTorch (Q3-b)	Thomas Koppens
OpenCV (Q4)	Sean D. Muscat

Project Overview

This project focuses on the use of data augmentation techniques with popular machine learning frameworks, including TensorFlow, PyTorch, and OpenCV. Data augmentation is an essential step in the preprocessing pipeline of machine learning models, particularly for image classification tasks, because it improves model generalization by artificially expanding the training dataset.

Question 3: Data Augmentation in Mainstream Machine Learning Frameworks

In Question 3, we look at how data augmentation is implemented with two popular machine learning (ML) frameworks: TensorFlow and PyTorch. The ImageDataGenerator class in TensorFlow allows for the application of a variety of image transformations. Similarly, PyTorch includes the transforms module, which enables advanced image manipulation techniques. By applying these tools to the COTS (Common Objects of a Traveling Scientist) dataset, we show how effective each framework is at generating augmented data. The task entails creating Jupyter Notebooks for both frameworks, and producing multiple augmented images per input. The results are then saved and analyzed using comparative metrics to determine the differences with the original images.

Question 4: Implementing Data Augmentation Techniques Using OpenCV

In Question 4, we look at how to implement data augmentation techniques with OpenCV. Unlike TensorFlow and PyTorch, OpenCV requires a more manual approach to transformations. This question addresses both photometric and geometric distortions, including brightness variation, contrast adjustment, rotation, saturation, shearing and translation. We use these techniques on the same subset of the COTS dataset as in Question 3, resulting in multiple augmented images per input. The goal is to compare OpenCV results to TensorFlow and PyTorch results, which will provide insights into the practical application and effectiveness of custom augmentation implementations.

---

q3-a

May 28, 2024

1 This JupyterNotebook was taken care by: `Andrea Filiberto Lucas (0279704L)`

1.1 Preamble

1.1.1 Setup and Library Importation

The provided code snippet bellow, creates the initial environment for a Jupyter Notebook session. It begins by importing modules required for the tasks ahead, such as `ImageDataGenerator` from `TensorFlow for data augmentation`, `os` for operating system interaction, `numpy` for numerical operations, `matplotlib.pyplot` for visualisation, and `shutil` for high-level file operations. It also imports `re` for regular expressions, `pandas` for data manipulation and analysis, and `seaborn` for statistical data visualisation, which expands the environment's functionality for a variety of data analysis tasks.

[52]:
```
# Import all the necessary modules
from tensorflow.keras.preprocessing.image import ImageDataGenerator # Import␣
↪ImageDataGenerator for data augmentation  # type: ignore
from PIL import Image  # Importing the Image module from the Python Imaging␣
↪Library (PIL) for image manipulation.
import os  # Import os module for interacting with the operating system
import numpy as np  # Import NumPy for numerical operations
import matplotlib.pyplot as plt  # Import matplotlib for visualization
import shutil  # Module for high-level file operations
import re  # Importing the 're' module for regular expressions.
import pandas as pd  # Importing the 'pandas' library and aliasing it as 'pd'␣
↪for convenience.
import seaborn as sns  # Importing the 'seaborn' library and aliasing it as␣
↪'sns' for convenience.
```

Furthermore, the `installer checker` block ensures that the necessary `libraries are installed locally`. It looks for `TensorFlow and scikit-image` (for ssim metric) and installs them if they're not available. This ensures that the notebook can operate seamlessly regardless of the user's local environment, allowing for a smooth and uninterrupted workflow.

[53]:
```
# Check if TensorFlow is installed, if not, install it
try:
    import tensorflow as tf  # Import TensorFlow library for deep learning tasks
```

```
except ImportError:
    !pip install tensorflow

# Check if scikit-image is installed, if not, install it
try:
    from skimage.metrics import structural_similarity as ssim  # Import SSIM␣
 ↪metric from scikit-image for image comparison
except ImportError:
    !pip install scikit-image
```

1.1.2 Setting Up Directory Structure

This section of code creates a `directory structure` for the project. It starts with defining a base directory, `base_dir`, which is set as the parent directory ('..'). Then it generates subdirectories for various purposes within the project. Specifically, it creates `three directories`: one for the COTS dataset (`cots_dataset_dir`), one for TensorFlow results (`results_base_dir`), and one for metrics (`metrics_dir`). Finally, it checks to see if these directories already exist and creates them if they do not.

[54]:
```
# Define base directory
base_dir = os.path.abspath('..')

# Define subdirectories
cots_dataset_dir = os.path.join(base_dir, '0. COTS_Img')
results_base_dir = os.path.join(base_dir, '3. Results', 'TensorFlow')
metrics_dir = os.path.join(results_base_dir, '0. Metrics')

# Create directories if they do not exist
try:
    for directory in [results_base_dir, metrics_dir]:
        os.makedirs(directory, exist_ok=True)
except OSError as e:
    print(f"Error creating directories: {e}")
```

1.1.3 Selected Images for COTS Dataset

This code snippet gets a list of image files from a given directory and displays them in a subplot layout. Each image is loaded, displayed (without axes), and identified by its filename, excluding the '.jpeg' extension. The resulting visualization provides an overview of the images selected from the COTS (Common Objects of a Traveling Scientist) dataset.

[55]:
```
# Get list of image files
image_files = [f for f in os.listdir(cots_dataset_dir) if f.endswith('.jpeg')]

# Display images
fig, axs = plt.subplots(1, len(image_files), figsize=(15, 5))
for i, image_file in enumerate(image_files):
```

```
    image_path = os.path.join(cots_dataset_dir, image_file)
    image = Image.open(image_path)
    axs[i].imshow(image)
    axs[i].axis('off')
    axs[i].set_title(image_file[:-5])  # Removing the '.jpeg' extension for␣
 ↪labeling

# Adjust spacing between title and images
plt.subplots_adjust(top=1.4)

plt.suptitle("Images selected for COTS dataset", fontsize=16)
plt.show()
```

1.2 Defining Data Augmentation Techniques

In the following segment, the code handles data augmentation strategies using TensorFlow's `ImageDataGenerator`. These techniques are critical for increasing the diversity and resilience of the training dataset. The dataset becomes more versatile by incorporating a variety of transformations such as `rotation`, `width and height shifting`, `shear`, `zooming`, and `horizontal flipping`. Each transformation specifies the degree of alteration, providing precise control over augmentation intensity. Finally, the `fill_mode` directive specifies the pixel interpolation strategy, which ensures that newly generated data is seamlessly integrated. This comprehensive approach ensures the creation of a diverse and robust dataset, which strengthens the efficacy of future machine learning efforts.

[56]:
```
# Define data augmentation transformations
datagen = ImageDataGenerator(
    rotation_range=20,              # Rotation
    width_shift_range=0.1,          # Width shift
    height_shift_range=0.1,         # Height shift
    shear_range=0.2,                # Shear
    zoom_range=0.2,                 # Zoom
    horizontal_flip=True,           # Horizontal flip
    fill_mode='nearest'             # Fill mode
)
```

1.3 Quantifying Structural Similarity

`Calculate_ssim` is an important function for determining the structural similarity of two images. Initially, it ensures that the input images are represented as `NumPy arrays` before converting them

to grayscale, which is required for comparative analysis. The RGB images are then converted to grayscale using TensorFlow's image processing capabilities. These grayscale representations are then converted into NumPy arrays for use in the `structural similarity (SSIM) metric` calculation. The SSIM score, calculated with the ssim function from the `scikit-image` library, quantifies the similarity of the grayscale images. The SSIM calculation is fine-tuned using parameters such as full and data_range, which allows for more accurate assessment. Finally, the function returns the SSIM score, which provides valuable information about the structural similarity of the input images. This process is used in a variety of computer vision applications, including image quality assessment, object recognition, and more. The higher the SSIM score, the closer the structural similarity between the two images.

[57]:
```
def calculate_ssim(image1, image2):
    # Ensure images are NumPy arrays and convert to grayscale
    image1_np = tf.keras.preprocessing.image.img_to_array(image1).
↪astype('float32') / 255.0
    image2_np = tf.keras.preprocessing.image.img_to_array(image2).
↪astype('float32') / 255.0

    # Convert RGB images to grayscale
    image1_gray = tf.image.rgb_to_grayscale(image1_np)
    image2_gray = tf.image.rgb_to_grayscale(image2_np)

    # Convert grayscale TensorFlow tensors to NumPy arrays
    image1_gray_np = image1_gray.numpy().squeeze()
    image2_gray_np = image2_gray.numpy().squeeze()

    # Calculate SSIM
    score, _ = ssim(image1_gray_np, image2_gray_np, full=True,␣
↪data_range=image2_gray_np.max() - image2_gray_np.min())
    return score
```

1.4 Data Augmentation & Metric Calculation

This extensive main section handles the augmentation of images from a given dataset while also computing comparative metrics. The script begins by initialising augmentation procedures for each image in the dataset, creating a directory structure to hold the augmented images and their metrics. It starts by loading the image, converting it to a NumPy array, and then saving the original image. Augmentation is then performed iteratively, yielding a series of modified images using the defined augmentation techniques. The script uses the previously defined `calculate_ssim` function to determine the structural similarity between each augmented image and the original. Both the augmented images and their associated SSIM metrics are saved in their respective folders (locally). In addition, the script creates a `comparative plot of the original and augmented images`, allowing for qualitative assessment. Each plot is saved as a reference. In addition, the SSIM metrics are aggregated and saved to a text file for further analysis. Finally, a `confirmation message is printed` to the console indicating that the augmented images and SSIM metrics were successfully saved, improving user interaction and workflow transparency.

[58]:
```
# Generate augmented images and calculate comparative metrics
metrics = []

for filename in os.listdir(cots_dataset_dir):
    img_path = os.path.join(cots_dataset_dir, filename)
    img = tf.keras.preprocessing.image.load_img(img_path)
    x = tf.keras.preprocessing.image.img_to_array(img)
    x = x.reshape((1,) + x.shape)
    original_image = x.squeeze(axis=0)

    # Create or clean the directory for each image's augmented set
    image_dir = os.path.join(results_base_dir, filename.split('.')[0])
    if os.path.exists(image_dir):
        shutil.rmtree(image_dir)  # Delete existing content
    os.makedirs(image_dir, exist_ok=True)

    # Save the original image
    original_image_path = os.path.join(image_dir, f'1. original_{filename}')
    tf.keras.preprocessing.image.save_img(original_image_path, original_image)

    augmented_images = []
    i = 1  # Start numbering from 1
    for batch in datagen.flow(x, batch_size=1):
        augmented_image = batch.squeeze(axis=0)
        ssim_score = calculate_ssim(original_image, augmented_image)
        metrics.append((filename, ssim_score))

        # Save the augmented image with the current value of i
        aug_filename = f"{filename.split('.')[0]}_augmented_{i}.jpeg"
        aug_image_path = os.path.join(image_dir, aug_filename)
        tf.keras.preprocessing.image.save_img(aug_image_path, augmented_image)

        augmented_images.append(aug_image_path)

        i += 1
        if i > 8:  # Stop after 8 augmented images
            break

    # Plot the original and augmented images
    fig, axes = plt.subplots(1, 9, figsize=(25, 5))
    axes[0].imshow(original_image.astype('uint8'))
    axes[0].set_title('Original')
    axes[0].axis('off')

    for j, aug_img_path in enumerate(augmented_images):
        aug_img = tf.keras.preprocessing.image.load_img(aug_img_path)
```

```
        aug_img = tf.keras.preprocessing.image.img_to_array(aug_img).
↪astype('uint8')
        axes[j + 1].imshow(aug_img)
        axes[j + 1].set_title(f'Augmented {j+1}')
        axes[j + 1].axis('off')

    # Save the plot
    plot_path = os.path.join(image_dir, '0. augmentation_cmp.png')
    plt.savefig(plot_path)
    plt.show()
    plt.close(fig)  # Close the figure to release memory

# Save SSIM metrics to a file
metrics_file = os.path.join(metrics_dir, '0. ssim_metrics.txt')

# Open the file in write mode
with open(metrics_file, 'w') as f:
    previous_filename = None
    augmentation_index = 1
    for filename, ssim_score in metrics:
        # Add a newline when transitioning to a new image
        if previous_filename and filename != previous_filename:
            f.write('\n')
            augmentation_index = 1  # Reset augmentation index for new image

        # Construct the augmented filename
        augmented_filename = f"{filename.split('.
↪')[0]}_augmented_{augmentation_index}.jpeg"

        # Write the SSIM metric for the current image
        f.write(f'{augmented_filename}: SSIM = {ssim_score:.4f}\n')

        previous_filename = filename
        augmentation_index += 1

# Printing a confirmation message (in bold and green)
print(f'\033[1;32mAugmented images successfully saved to␣
↪{results_base_dir}\033[0m')
print(f'\033[1;32mSSIM metrics successfully saved to {metrics_file}\033[0m')
```

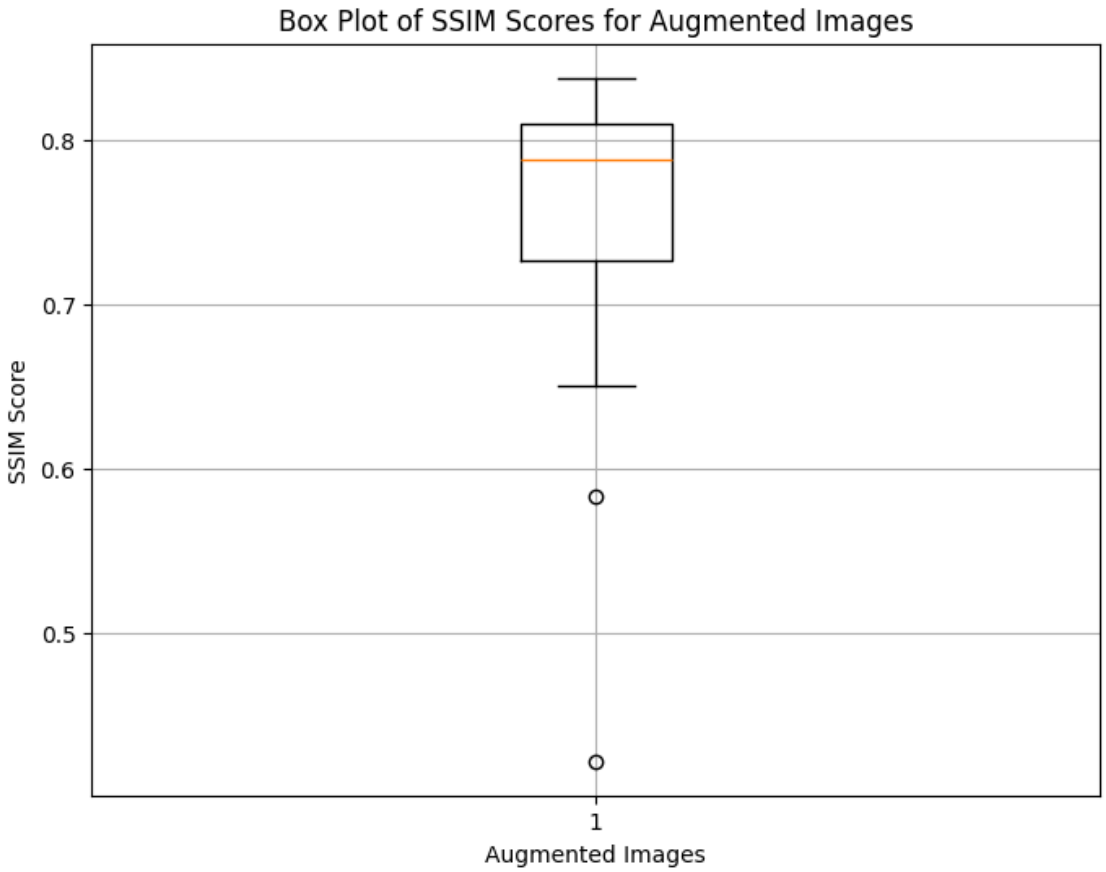
```
Augmented images successfully saved to /Users/afl/Documents/Uni/Year
2/Lectures/SEM2/CV/Group Project/Part B/Question 3/3. Results/TensorFlow
SSIM metrics successfully saved to /Users/afl/Documents/Uni/Year
2/Lectures/SEM2/CV/Group Project/Part B/Question 3/3. Results/TensorFlow/0.
Metrics/0. ssim_metrics.txt
```

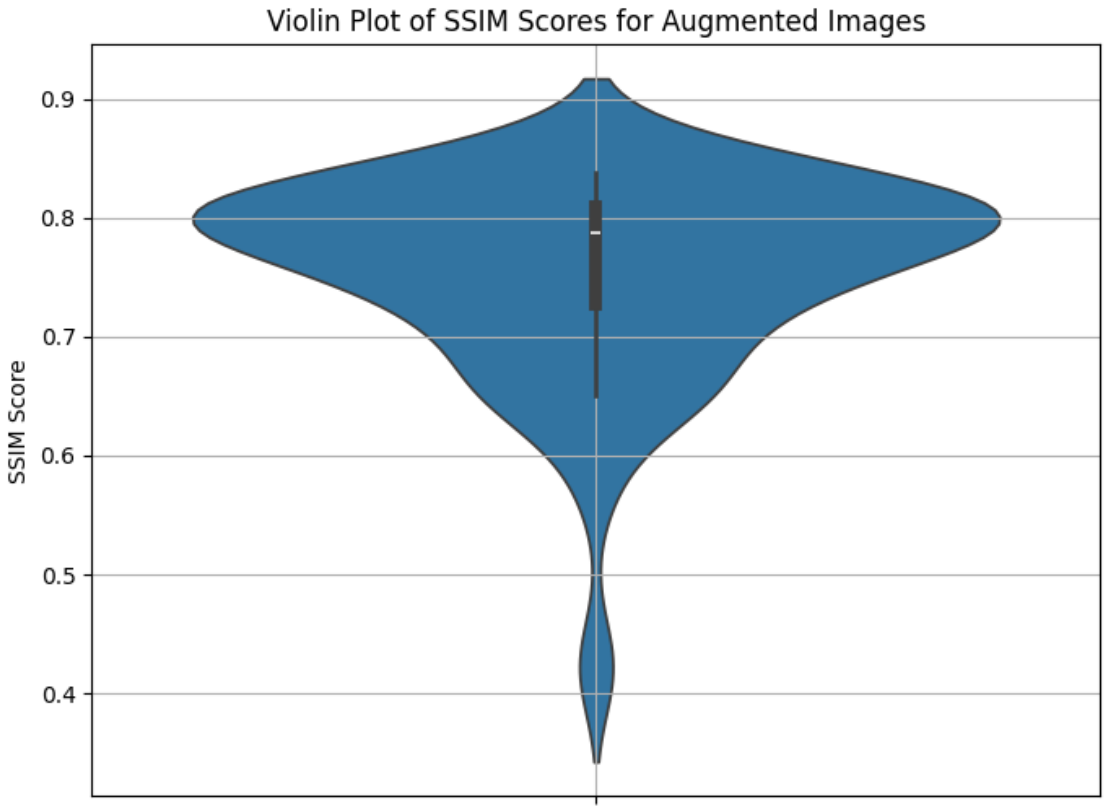
1.5 Visualizing SSIM Scores

This final code block examines the `SSIM scores` for augmented images. Initially, it loads SSIM metrics from a specified file using `pandas`, parsing filenames and SSIM scores separated by colons. It then extracts the numeric portion of SSIM scores from the 'SSIM' column using a regex pattern. Following data preparation, three different visualization techniques are used to understand the distribution of SSIM scores. First, a `box plot` is created to depict `quartiles and outliers`. Then, a `violin plot`, which combines a `box plot` and a `kernel density plot`, provides a complete picture of the distribution. Finally, a `histogram shows the frequency distribution of SSIM scores` in predefined bins. The resulting figures are displayed with the `matplotlib` and `seaborn` libraries.

[59]:
```
# Load SSIM metrics from the specified file using pandas
metrics_df = pd.read_csv(metrics_file, sep=':', header=None, names=['Filename',␣
↪'SSIM'])

# Extract numerical part of SSIM scores from the 'SSIM' column
metrics_df['SSIM'] = metrics_df['SSIM'].apply(lambda x: float(re.search(r'\d+\.
↪\d+', x).group()))
```

```python
# Visualize the distribution of SSIM scores using a box plot
plt.figure(figsize=(8, 6))
plt.boxplot(metrics_df['SSIM'])
plt.title('Box Plot of SSIM Scores for Augmented Images')
plt.xlabel('Augmented Images')
plt.ylabel('SSIM Score')
plt.grid(True)
plt.savefig(os.path.join(metrics_dir, 'box_plot_ssim.png'))  # Save the box plot
plt.show()

# Visualize the distribution of SSIM scores using a violin plot
plt.figure(figsize=(8, 6))
sns.violinplot(y=metrics_df['SSIM'])
plt.title('Violin Plot of SSIM Scores for Augmented Images')
plt.ylabel('SSIM Score')
plt.grid(True)
plt.savefig(os.path.join(metrics_dir, 'violin_plot_ssim.png'))  # Save the␣
 ↪violin plot
plt.show()

# Visualize the distribution of SSIM scores using a histogram
plt.figure(figsize=(8, 6))
plt.hist(metrics_df['SSIM'], bins=10, edgecolor='k')
plt.title('Histogram of SSIM Scores for Augmented Images')
plt.xlabel('SSIM Score')
plt.ylabel('Frequency')
plt.grid(True)
plt.savefig(os.path.join(metrics_dir, 'histogram_ssim.png'))  # Save the␣
 ↪histogram
plt.show()
```


Box Plot of SSIM Scores for Augmented Images
SSIM Score
0.8
0.7
0.6
0.5
1
Augmented Images


Violin Plot of SSIM Scores for Augmented Images
SSIM Score
0.9
0.8
0.7
0.6
0.5
0.4

The above three visualisations provide a thorough examination of SSIM scores for augmented images.

The first box plot, summarises the SSIM score distribution by emphasising the median, quartiles, and possible outliers. This plot shows that most SSIM scores are clustered around 0.8, with a few lower outliers around 0.5, indicating a high degree of similarity in the majority of augmented images, though there are some exceptions.

The second plot, a violin plot, combines elements of a box plot and a density plot to depict the distribution's shape and spread. This plot emphasises that the SSIM scores are densely packed around 0.8, with a smooth distribution that tapers off towards lower scores, demonstrating both the concentration and variability of the scores.

The third plot, a histogram, shows the frequency distribution of the SSIM scores. The majority of scores fall between 0.7 and 0.8, with fewer images scoring less than 0.7. This histogram confirms the findings from the previous plots, indicating that while the majority of augmented images maintain high structural similarity, a small subset exhibits significant deviation.

q3-b

June 7, 2024

1 This JupyterNotebook was taken care by: `Thomas Koppens` (029204L)

1.0.1 Preamble

Most of the code between the TensorFlow and PyTorch solutions is shared since, naturally, only the augmentation and I/O are different.

Imports In the PyTorch version of this question, torchvision's `transforms` are used. In Particular, v2 is used since it implements more transformations and is generally quicker.

Additionally, PyTorch Ignite's implementation of SSIM is used for evaluation.

[16]:
```
# Check if torch is installed, if not, install it
try:
    from torchvision.transforms import v2
except ImportError:
    !pip install torch

# Check if ignite.metrics is installed, if not, install it
try:
    from ignite.metrics import SSIM
except ImportError:
    !pip install pytorch-ignite
```

[10]:
```
# Data augmentation related imports
from torchvision.io import read_image, write_jpeg
import numpy as np

# Directory related imports
import os
import shutil

# Plot related imports
import pandas as pd
import re
import matplotlib.pyplot as plt
```

Directories The directories used are identical to the TensorFlow solution, with a `PyTorch` directory being used instead of `TensorFlow`.

[11]:
```
base_dir = os.path.abspath("..")
cots_dataset_dir = os.path.join(base_dir, "0. COTS_Img")
results_base_dir = os.path.join(base_dir, "3. Results", "PyTorch")
metrics_dir = os.path.join(results_base_dir, "0. Metrics")
```

Data Augmentation The `transforms.v2` module defines a number of data augmentation techniques. These include both individual techniques as well as predefined data augmentation solutions such as RandAugment.

For the purposes of this project, only individual techniques are used. Furthermore, a mixture of geometric and photometric techniques are used to produce highly dissimilar images whilst retaining the semantics of the object in the image.

Geometric Transformations

1. `Rotation` - an image can be rotated by a maximum of 15°. The highest quality interpolation mode, bilinear, was used.
2. `RandomResizedCrop` - the final image resolution is restored to 1820x720. The minimum scale is half the original image.
3. `RandomHorizontalFlip` - there is a 50% chance the image will be flipped. Vertical flipping was not implemented since the object in the image loses semantic value.
4. `ColorJitter` - this really implements 3 transforms: brightness, saturation, and contrast. The `hue` parameter is set to 0 such that there is no hue distortion, since it is more likely to affect semantics.
5. `GaussianBlur` - applies a minor blur to the image since small values of sigma were used.

[12]:
```
transforms = v2.Compose([
    # Maximum rotation of 15deg in either direction
    v2.RandomRotation(15, interpolation=v2.InterpolationMode.BILINEAR),
    v2.RandomResizedCrop(size=(720, 1280),
                         scale=(0.5, 1),
                         interpolation=v2.InterpolationMode.BILINEAR),
    # 50% chance of occurring
    v2.RandomHorizontalFlip(),
    v2.ColorJitter(
        brightness=0.3,
        contrast=0.2,
        saturation=0.5,
        hue=0
    ),
    v2.GaussianBlur(kernel_size=3, sigma=(0.01, 0.1))
])
```

Evaluation Metric As in the previous version, SSIM was used to evaluate the similarity between the original and augmented images.

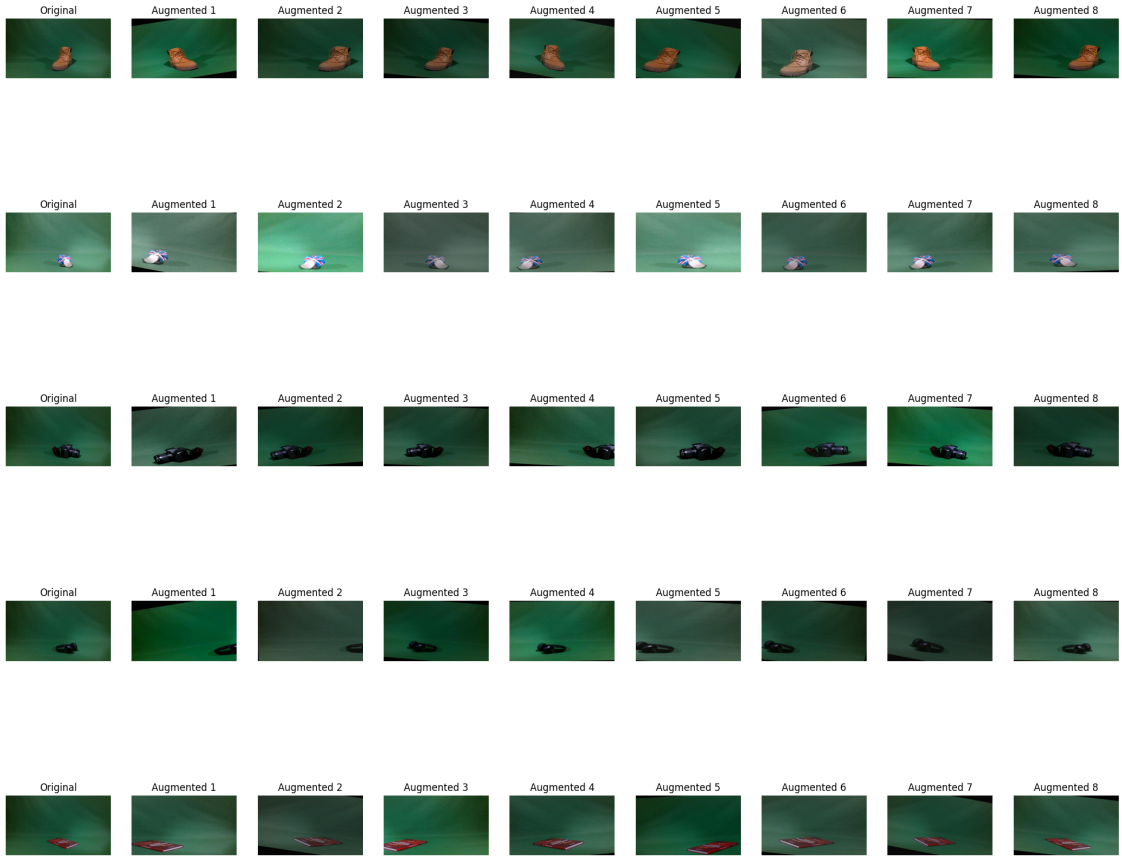
In PyTorch Ignite, an SSIM metric can be created by defining the data range. In the case of RGB image tensors, this value is 255.

[13]:
```
def calculate_ssim(image1, image2):
    # Adding required batch dimension
    image1 = image1.unsqueeze(0)
    image2 = image2.unsqueeze(0)

    # Creating SSIM metric
    ssim_metric = SSIM(data_range=255)
    ssim_metric.update((image1, image2))

    # Calculate SSIM and return its value. Setting a minimum of 0 for␣
 ↪calculation errors
    return ssim_metric.compute()
```

Calculations This process is effectively identical to the TensorFlow implementation, only loading, saving, transforming, and evalulating images using PyTorch's tools instead.

[14]:
```
metrics = []

for filename in os.listdir(cots_dataset_dir):
    img_path = os.path.join(cots_dataset_dir, filename)
    original_image = read_image(img_path)

    # Create or clean the directory for each image"s augmented set
    image_dir = os.path.join(results_base_dir, filename.split(".")[0])
    if os.path.exists(image_dir):
        shutil.rmtree(image_dir) # Delete existing content
    os.makedirs(image_dir, exist_ok=True)

    # Save the original image
    original_image_path = os.path.join(image_dir, f"1. original_{filename}")
    write_jpeg(original_image, original_image_path)

    augmented_images = []
    for i in range(1, 9):
        augmented_image = transforms(original_image)
        ssim_score = calculate_ssim(original_image, augmented_image)
        metrics.append((filename, ssim_score))

        # Save the augmented image with the current value of i
        aug_filename = f"{filename.split('.')[0]}_augmented_{i}.jpeg"
        aug_image_path = os.path.join(image_dir, aug_filename)
        write_jpeg(augmented_image, aug_image_path)

        augmented_images.append(aug_image_path)
```

```
    # Plot the original and augmented images
    fig, axes = plt.subplots(1, 9, figsize=(25, 5))
    axes[0].imshow(original_image.permute(1, 2, 0))
    axes[0].set_title("Original")
    axes[0].axis("off")

    for j, aug_img_path in enumerate(augmented_images):
        aug_img = read_image(aug_img_path)
        axes[j + 1].imshow(aug_img.permute(1, 2, 0))
        axes[j + 1].set_title(f"Augmented {j+1}")
        axes[j + 1].axis("off")


    # Save the plot
    plot_path = os.path.join(image_dir, "0. augmentation_cmp.png")
    plt.savefig(plot_path)
    plt.show()
    plt.close(fig)  # Close the figure to release memory

# Save SSIM metrics to a file
metrics_file = os.path.join(metrics_dir, "0. ssim_metrics.txt")

# Open the file in write mode
with open(metrics_file, "w") as f:
    previous_filename = None
    augmentation_index = 1
    for filename, ssim_score in metrics:
        # Add a newline when transitioning to a new image
        if previous_filename and filename != previous_filename:
            f.write("\n")
            augmentation_index = 1  # Reset augmentation index for new image

        # Construct the augmented filename
        augmented_filename = f"{filename.split('.
 ↪')[0]}_augmented_{augmentation_index}.jpeg"

        # Write the SSIM metric for the current image
        f.write(f"{augmented_filename}: SSIM = {ssim_score:.4f}\n")

        previous_filename = filename
        augmentation_index += 1

# Printing a confirmation message (in bold and green)
print(f"\033[1;32mAugmented images successfully saved to␣
 ↪{results_base_dir}\033[0m")
print(f"\033[1;32mSSIM metrics successfully saved to {metrics_file}\033[0m")
```

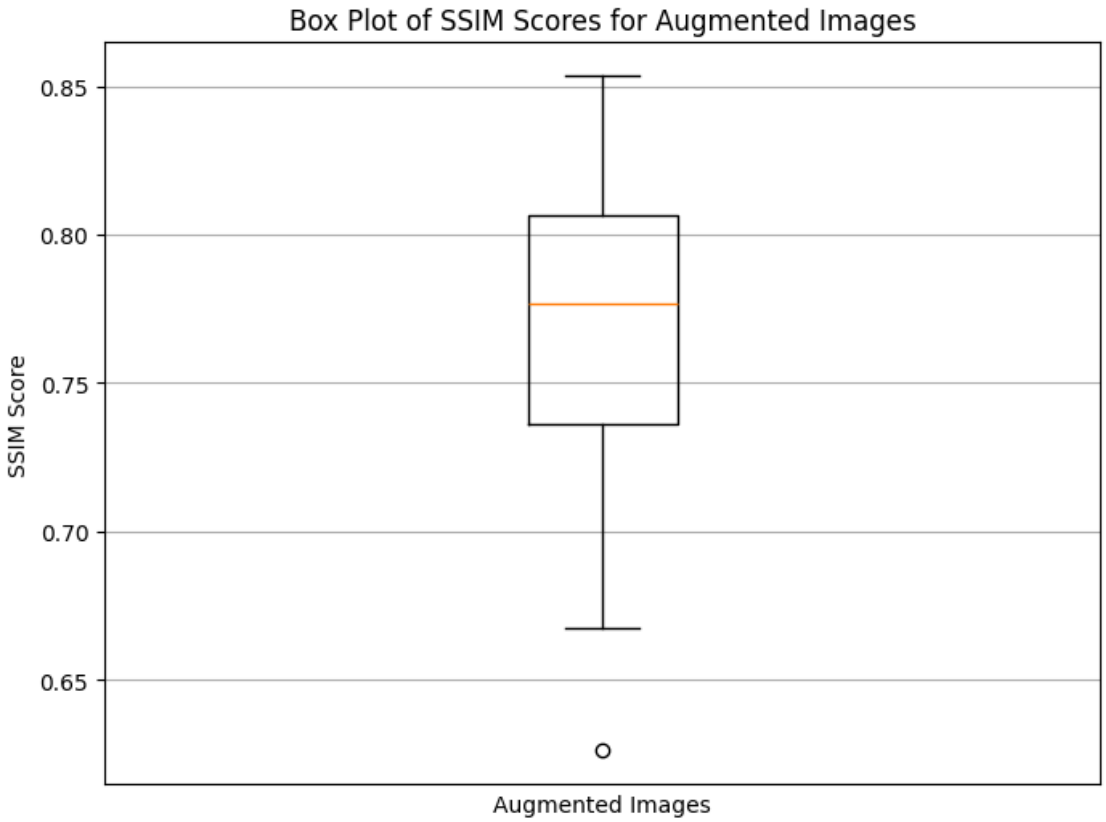


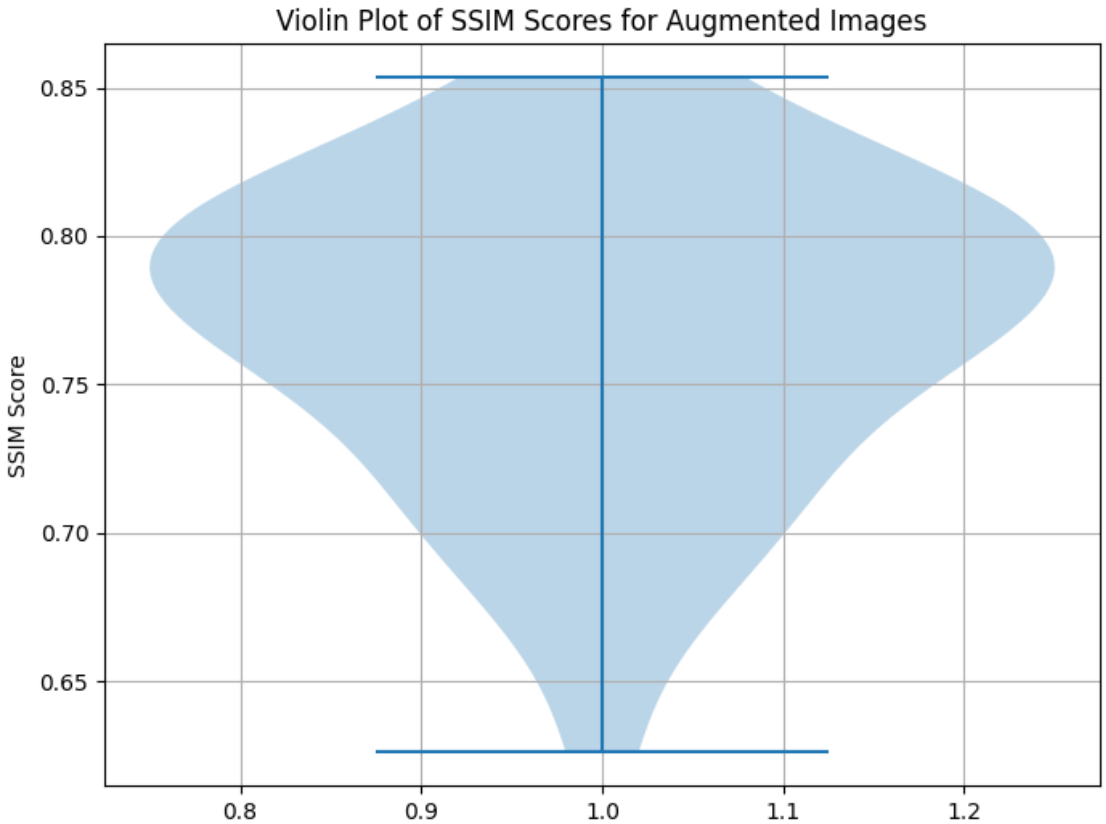

```
Augmented images successfully saved to /Users/afl/Documents/Uni/Year
2/Lectures/SEM2/CV/Group Project/Part B/Question 3/3. Results/PyTorch
SSIM metrics successfully saved to /Users/afl/Documents/Uni/Year
2/Lectures/SEM2/CV/Group Project/Part B/Question 3/3. Results/PyTorch/0.
Metrics/0. ssim_metrics.txt
```

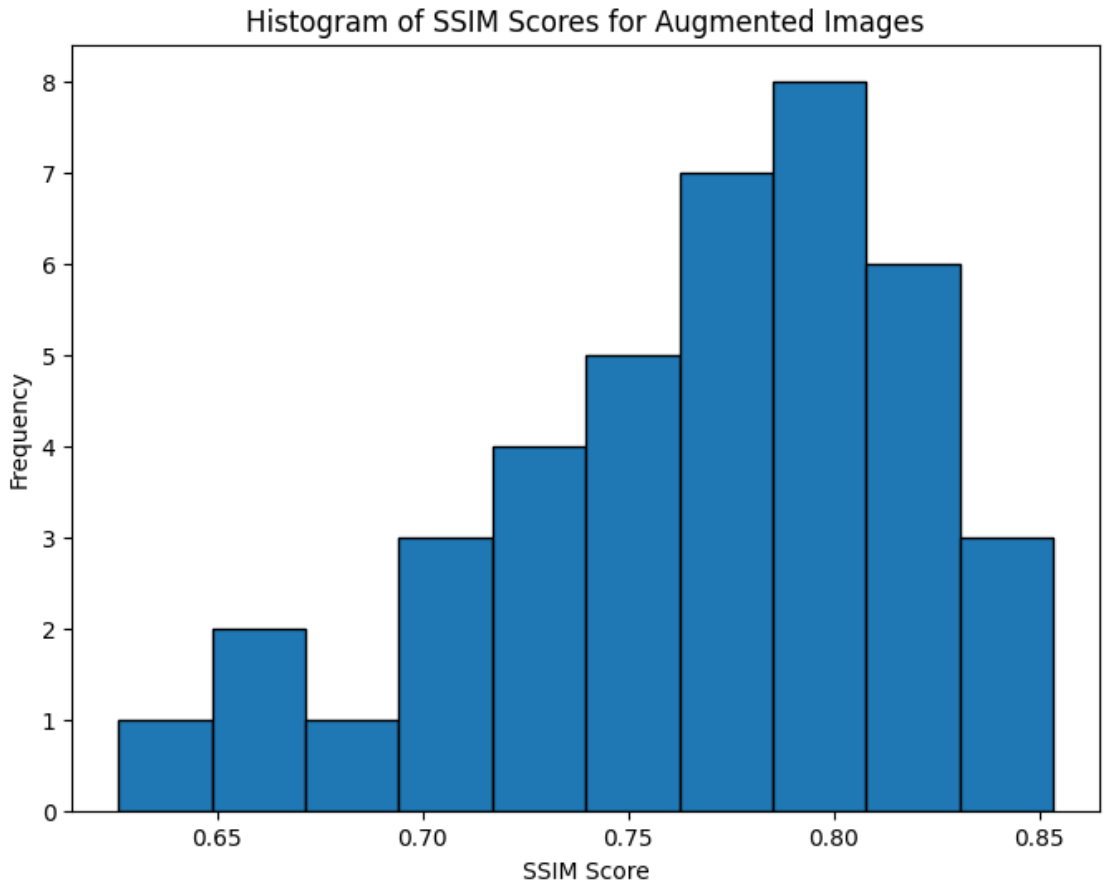
Evaluation The same plots were used to visualise the similarity of the augmented images with respect to their original image. The combined use of geometric and photometric transformations seems to have similar results to those used in the TensorFlow implementation, with a slight decrease in the average similarity and a tighter spread with no outliers.

[15]:
```
# Load SSIM metrics from the specified file using pandas
metrics_df = pd.read_csv(metrics_file, sep=":", header=None, names=["Filename",␣
↪"SSIM"])

# Extract numerical part of SSIM scores from the "SSIM" column
metrics_df["SSIM"] = metrics_df["SSIM"].apply(lambda x: float(re.search(r"\d+\.
↪\d+", x).group()))
```

```python
# Visualize the distribution of SSIM scores using a box plot
plt.figure(figsize=(8, 6))
plt.boxplot(metrics_df["SSIM"])
plt.title("Box Plot of SSIM Scores for Augmented Images")
plt.xlabel("Augmented Images")
plt.ylabel("SSIM Score")
plt.grid(True)
plt.savefig(os.path.join(metrics_dir, "box_plot_ssim.png"))  # Save the box plot
plt.xticks([])
plt.show()

# Visualize the distribution of SSIM scores using a violin plot
plt.figure(figsize=(8, 6))
plt.violinplot(metrics_df["SSIM"])
plt.title("Violin Plot of SSIM Scores for Augmented Images")
plt.ylabel("SSIM Score")
plt.grid(True)
plt.savefig(os.path.join(metrics_dir, "violin_plot_ssim.png"))  # Save the␣
↪violin plot
plt.show()

# Visualize the distribution of SSIM scores using a histogram
plt.figure(figsize=(8, 6))
plt.hist(metrics_df["SSIM"], bins=10, edgecolor="k")
plt.title("Histogram of SSIM Scores for Augmented Images")
plt.xlabel("SSIM Score")
plt.ylabel("Frequency")
plt.savefig(os.path.join(metrics_dir, "histogram_ssim.png"))  # Save the␣
↪histogram
plt.show()
```


Box Plot of SSIM Scores for Augmented Images
0.85
0.80
0.75
0.70
0.65
SSIM Score
Augmented Images


Violin Plot of SSIM Scores for Augmented Images
SSIM Score
0.85
0.80
0.75
0.70
0.65
0.8
0.9
1.0
1.1
1.2


Histogram of SSIM Scores for Augmented Images
Frequency
SSIM Score
0
1
2
3
4
5
6
7
8
0.65
0.70
0.75
0.80
0.85

q4

May 28, 2024

1 This JupyterNotebook was taken care by: `Sean David Muscat` (0172004L)

Preamble:

This code imports essential modules for the task, including os for directory operations, cv2 for image processing, matplotlib.pyplot for plotting, and numpy for numerical operations.

[14]:
```
# Importing all needed modules for this task
import os
import cv2
import matplotlib.pyplot as plt
import numpy as np
```

This code retrieves the current working directory and creates absolute paths to the image and results folders. The image folder is one level higher than the current directory, under '0. COTS_Img', and the results folder is also one level higher, under '2. Results'.

[15]:
```
# Get the current working directory (where the notebook is running)
base_dir = os.getcwd()
# Define the path to the image folder relative to the current directory
image_folder = os.path.abspath(os.path.join(base_dir, '..', '0. COTS_Img'))
# Define the path to the results folder relative to the current directory
results_folder = os.path.abspath(os.path.join(base_dir, '..', '2. Results'))
```

This code dynamically lists all JPEG image filenames in the COTS_Img folder and loads them with OpenCV. It converts each image to RGB format and saves it in a list. If an image fails to load, it displays an error message. If the images are successfully loaded, Matplotlib displays them in a single row of subplots, each labelled with the corresponding filename.

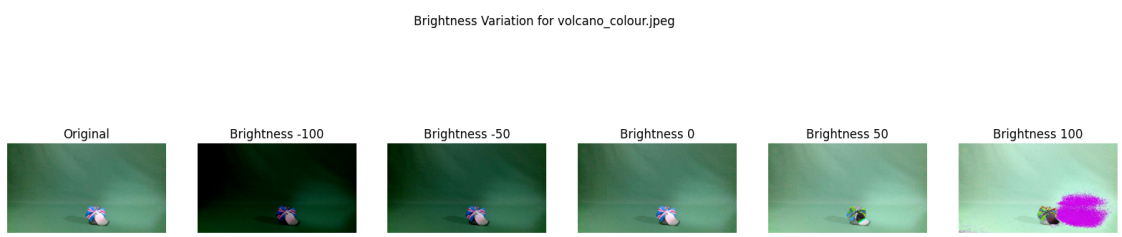
[16]:
```
# Dynamically list image filenames in the folder
image_files = [f for f in os.listdir(image_folder) if f.endswith(('.jpeg'))]

# Load images
images = []
for filename in image_files:
    img_path = os.path.join(image_folder, filename)
    img = cv2.imread(img_path)
    if img is not None:
```

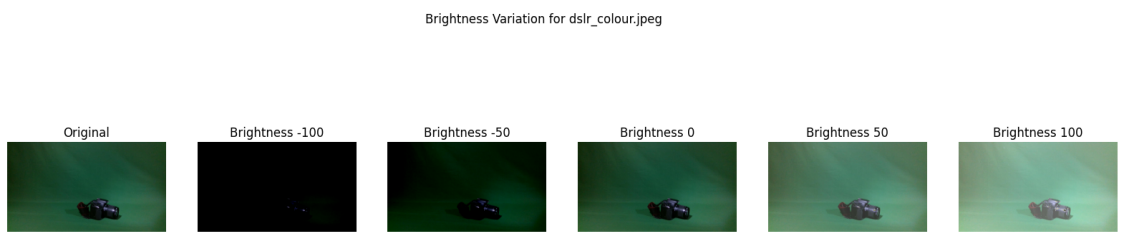
```
        img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        images.append((filename, img_rgb))
    else:
        print(f"Failed to load {filename} at path {img_path}")

# Check if images were loaded
if not images:
    print("No images were loaded. Please check the paths and filenames.")
else:
    # Display images
    fig, axes = plt.subplots(1, len(images), figsize=(20, 5))
    for ax, (filename, img) in zip(axes, images):
        ax.imshow(img)
        ax.set_title(filename)
        ax.axis('off')
    plt.show()
```

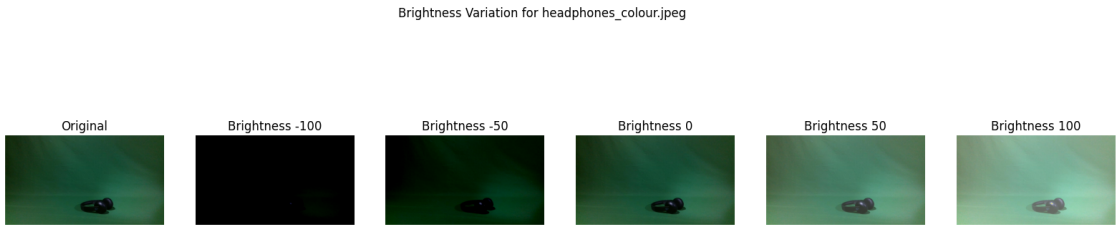
Brightness variation:

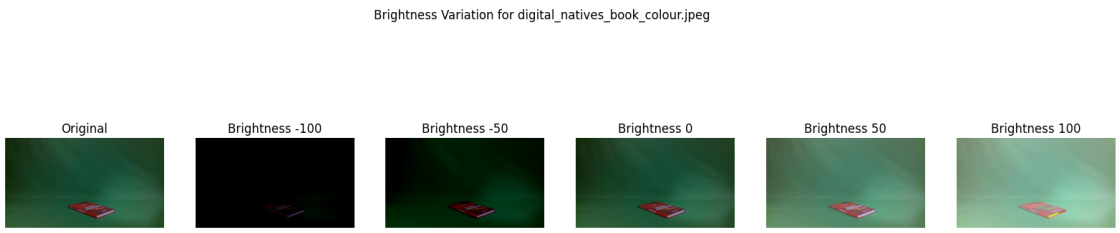
[17]:
```
# Ensure the 'Brightness' directory exists in the 'Results' folder
os.makedirs(os.path.join(results_folder, 'Brightness'), exist_ok=True)

# Function to adjust brightness
def adjust_brightness(image, brightness):
    # Clip the values to be in the valid range [0, 255]
    bright_img = np.clip(image + brightness, 0, 255).astype(np.uint8)
    return bright_img

# Brightness levels to apply
brightness_levels = [-100, -50, 0, 50, 100]

# Generate and display augmented images
for image_file, img in images:
    fig, axes = plt.subplots(1, len(brightness_levels) + 1, figsize=(20, 5))
    fig.suptitle(f'Brightness Variation for {image_file}')

    # Display original image
    axes[0].imshow(img)
    axes[0].set_title('Original')
    axes[0].axis('off')
```

```python
# Generate augmented images
for ax, level in zip(axes[1:], brightness_levels):
    augmented_img = adjust_brightness(img, level)
    ax.imshow(augmented_img)
    ax.set_title(f'Brightness {level}')
    ax.axis('off')

    # Save the augmented image in the 'Brightness' directory within the␣
↪'Results' folder
    output_filename = os.path.join(results_folder, 'Brightness',␣
↪f'brightness_{level}_{image_file}')
    plt.imsave(output_filename, augmented_img)

plt.show()
```









The code first makes sure that the augmented photographs will be kept in a directory called "brightness." Then, to change the brightness levels of images, the adjust_brightness() function is defined. It accepts a picture and a brightness parameter as inputs, adds the given value to each pixel to modify brightness, and makes sure the values of the pixels stay within the allowed range of 0 to 255. The modified image is returned by the function. The next step is to establish a list of brightness levels, which go from -100 to 100. The brightness variations that will be applied to the photos are represented by this list.

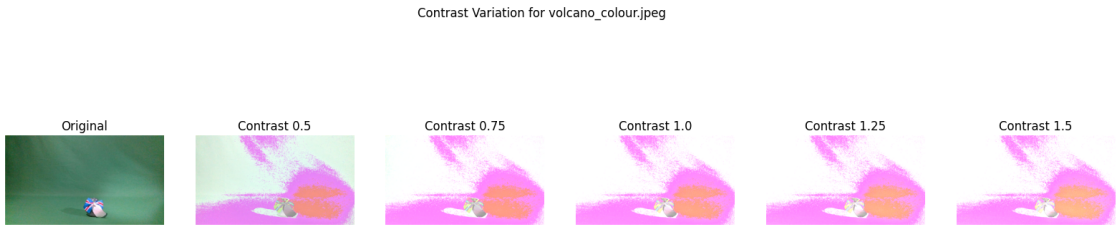
Lastly, a collection of augmented images is created for each image by modifying the brightness levels in accordance with the specified list. These enhanced pictures are shown next to the original picture so that various brightness levels can be visually compared.

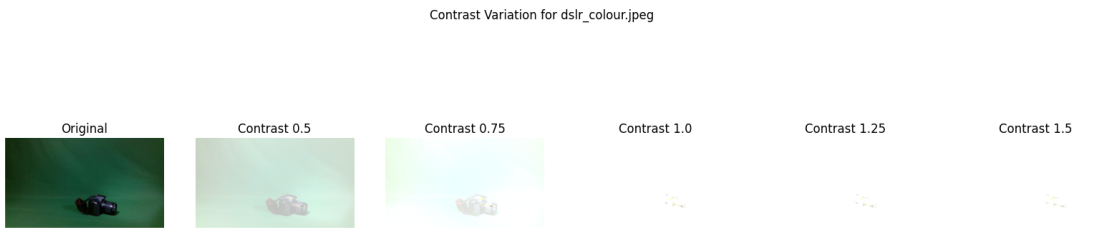
The final adjustments to the brightness levels, for example, will have an impact on how the photographs appear visually. Brightness will increase with positive numbers and decrease with negative ones. The image will remain unaltered if the value is 0. The magnitude of the value determines how much of a change there is. Changes will be more noticeable at higher levels, and less noticeable at lower values. It is possible to fine-tune the brightness of photographs to obtain desired visual effects by experimenting with different values.

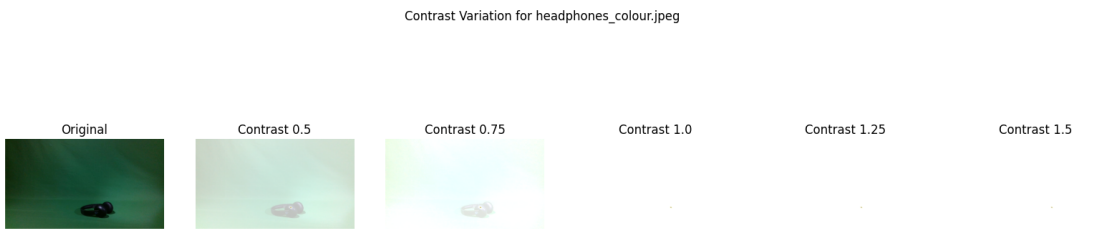
Contrast Variation:

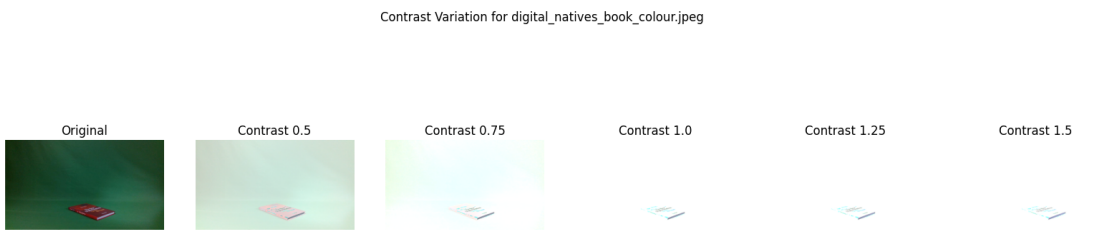
[18]:
```
# Ensure the 'Contrast' directory exists in the 'Results' folder
os.makedirs(os.path.join(results_folder, 'Contrast'), exist_ok=True)

# Function to adjust contrast
def adjust_contrast(image, contrast):
    # Calculate the midpoint value
    midpoint = 128
    # Apply contrast adjustment
```

```
    contrast_img = np.clip(midpoint + contrast * (image - midpoint), 0, 255).
 ↪astype(np.uint8)
    return contrast_img

# Contrast levels to apply
contrast_levels = [0.5, 0.75, 1.0, 1.25, 1.5]

# Generate and display augmented images
for image_file, img in images:
    fig, axes = plt.subplots(1, len(contrast_levels) + 1, figsize=(20, 5))
    fig.suptitle(f'Contrast Variation for {image_file}')

    # Display original image
    axes[0].imshow(img)
    axes[0].set_title('Original')
    axes[0].axis('off')

    # Generate augmented images
    for ax, level in zip(axes[1:], contrast_levels):
        augmented_img = adjust_contrast(img, level)
        ax.imshow(augmented_img)
        ax.set_title(f'Contrast {level}')
        ax.axis('off')

        # Save the augmented image in the 'Contrast' directory within the␣
 ↪'Results' folder
        output_filename = os.path.join(results_folder, 'Contrast',␣
 ↪f'contrast_{level}_{image_file}')
        plt.imsave(output_filename, augmented_img)

    plt.show()
```









The code starts by ensuring that augmented photographs will be stored in a directory called "contrast." Then, it defines the adjust_contrast() function to alter the contrast levels of images. This function accepts an image and a contrast parameter as inputs. It calculates the midpoint value (which is set to 128) and applies a contrast adjustment formula to each pixel. The formula enhances or reduces contrast based on the given contrast parameter. The modified image is then returned by the function. Following this, a list of contrast levels ranging from 0.5 to 1.5 is established. These levels represent the variations in contrast that will be applied to the photos.

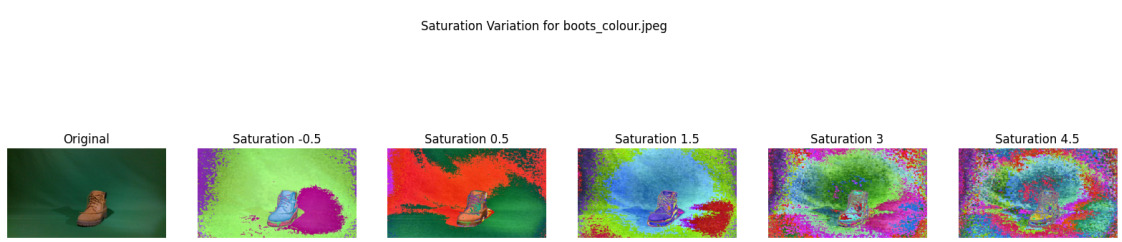
Lastly, a set of augmented images is generated for each image by adjusting the contrast levels according to the specified list. These enhanced images are displayed alongside the original image,

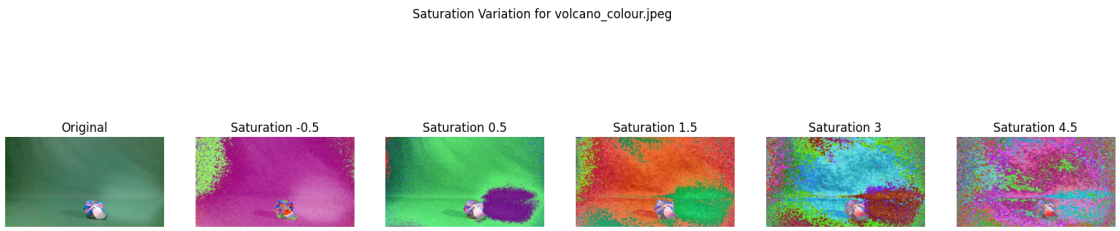
allowing for visual comparison of different contrast levels. The higher the contrast value given, the more "distorted" the image appears and the more certain colours (like the large pink found for the volcano object) pop out.

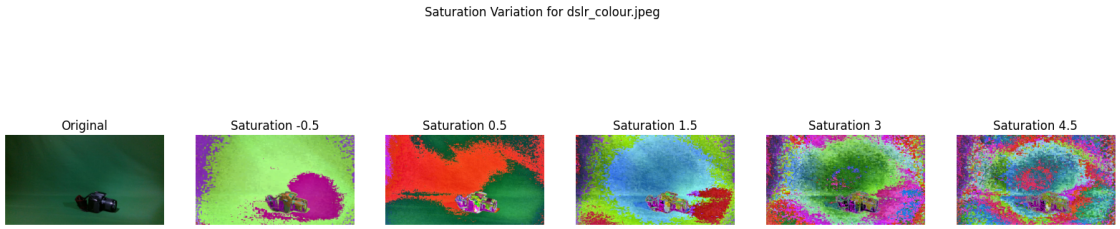
Saturation Variation:

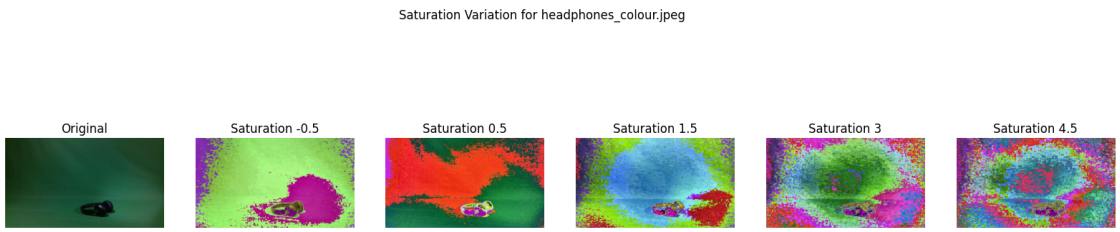
[19]:
```
# Ensure the 'Saturation' directory exists in the 'Results' folder
os.makedirs(os.path.join(results_folder, 'Saturation'), exist_ok=True)

# Function to adjust saturation using first principles
def adjust_saturation(image, saturation_factor):
    saturated_img = np.zeros_like(image)

    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            # Convert RGB to HSV
            r, g, b = image[i, j] / 255.0
            cmax = max(r, g, b)
            cmin = min(r, g, b)
            delta = cmax - cmin

            # Compute saturation value
            if cmax != 0:
                s = delta / cmax
            else:
                s = 0

            # Adjust saturation
            s *= saturation_factor

            if s == 0:
                r, g, b = 0, 0, 0
            else:
                # Compute RGB values
                h = 60 * (((r - g) / delta) % 6) if cmax == r else \
                    60 * (((g - b) / delta) + 2) if cmax == g else \
                    60 * (((b - r) / delta) + 4)
                x = s * (1 - abs((h / 60) % 2 - 1))
                m = cmax - s

                if 0 <= h < 60:
                    r, g, b = cmax, x, 0
                elif 60 <= h < 120:
                    r, g, b = x, cmax, 0
                elif 120 <= h < 180:
                    r, g, b = 0, cmax, x
                elif 180 <= h < 240:
                    r, g, b = 0, x, cmax
```

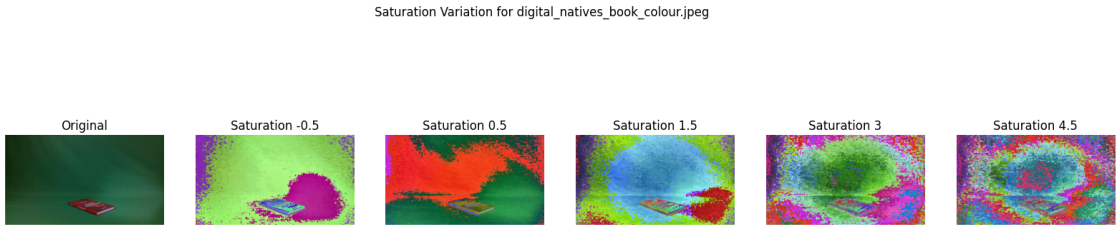
```
                elif 240 <= h < 300:
                    r, g, b = x, 0, cmax
                elif 300 <= h < 360:
                    r, g, b = cmax, 0, x

                r, g, b = (r + m) * 255, (g + m) * 255, (b + m) * 255

            saturated_img[i, j] = [r, g, b]

    return saturated_img.astype(np.uint8)

# Saturation levels to apply
saturation_levels = [-0.5, 0.5, 1.5, 3, 4.5]

# Generate and display augmented images
for image_file, img in images:
    fig, axes = plt.subplots(1, len(saturation_levels) + 1, figsize=(20, 5))
    fig.suptitle(f'Saturation Variation for {image_file}')

    # Display original image
    axes[0].imshow(img)
    axes[0].set_title('Original')
    axes[0].axis('off')

    # Generate augmented images
    for ax, level in zip(axes[1:], saturation_levels):
        augmented_img = adjust_saturation(img, level)
        ax.imshow(augmented_img)
        ax.set_title(f'Saturation {level}')
        ax.axis('off')

        # Save the augmented image in the 'Saturation' directory within the␣
↪'Results' folder
        output_filename = os.path.join(results_folder, 'Saturation',␣
↪f'saturation_{level}_{image_file}')
        plt.imsave(output_filename, augmented_img)

    plt.show()
```











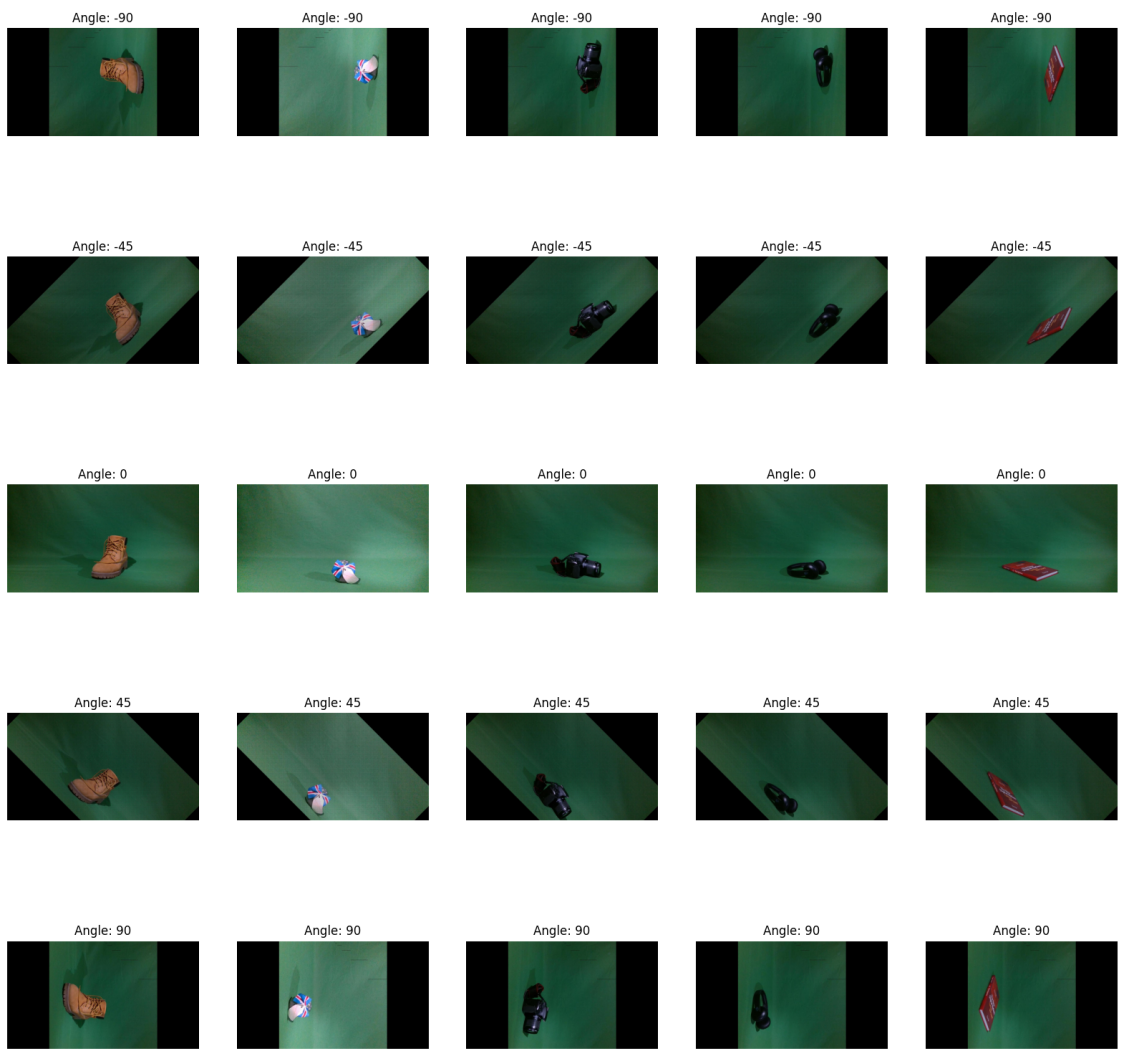
The code starts by ensuring that augmented photographs will be stored in a directory called "saturation." Then, it defines the adjust_saturation() function, which adjusts the saturation of images based on the given saturation factor. This function operates on the RGB color space and computes the saturation of each pixel using first principles of color theory. It converts the RGB values to HSV (Hue, Saturation, Value) color space, adjusts the saturation component, and then converts back to RGB. The modified image is returned by the function. Following this, a list of saturation levels ranging from -0.5 to 4.5 is established. These levels represent the variations in saturation that will be applied to the photos.

Lastly, a set of augmented images is generated for each image by adjusting the saturation levels according to the specified list. These enhanced images are displayed alongside the original image, allowing for visual comparison of different saturation levels.

Rotation:

[20]:
```
# Ensure the 'Rotation' directory exists in the 'Results' folder
os.makedirs(os.path.join(results_folder, 'Rotation'), exist_ok=True)

# Define function to rotate image using first principles
def rotate_image(image, angle):
    angle_rad = np.deg2rad(angle)
    rows, cols, _ = image.shape
    center_row, center_col = rows // 2, cols // 2

    rotated_image = np.zeros_like(image)

    for i in range(rows):
        for j in range(cols):
            y = i - center_row
            x = j - center_col

            new_x = int(center_col + x * np.cos(angle_rad) - y * np.
 ↪sin(angle_rad))
            new_y = int(center_row + x * np.sin(angle_rad) + y * np.
 ↪cos(angle_rad))

            if 0 <= new_x < cols and 0 <= new_y < rows:
                rotated_image[new_y, new_x] = image[i, j]

    return rotated_image

# Rotation angles to apply
angles = [-90, -45, 0, 45, 90]

# Generate augmented images with rotation
for image_file, img in images:
    for angle in angles:
        rotated_img = rotate_image(img, angle)

        # Save the rotated image in the 'Rotation' directory within the␣
 ↪'Results' folder
        output_filename = os.path.join(results_folder, 'Rotation',␣
 ↪f'rotation_{angle}_{image_file}')
        plt.imsave(output_filename, rotated_img)

# Display the augmented images
fig, axes = plt.subplots(len(angles), len(images), figsize=(20, 20))
```

```
for i, angle in enumerate(angles):
    for j, (image_file, _) in enumerate(images):
        ax = axes[i, j]
        ax.imshow(plt.imread(os.path.join(results_folder, 'Rotation',␣
 ↪f'rotation_{angle}_{image_file}')))
        ax.set_title(f"Angle: {angle}")
        ax.axis('off')

plt.show()
```



The code ensures that augmented photographs will be stored in a directory called "rotation." Then, it defines the rotate_image() function, which rotates images based on the specified angle using first principles of geometry. This function rotates each pixel of the image around its center by the given angle in degrees. It calculates the new coordinates of each pixel after rotation and assigns the pixel

value to the corresponding location in the rotated image. The rotated image is returned by the function. Following this, a list of rotation angles ranging from -90 to 90 degrees is established.

Next, a set of augmented images is generated for each image by applying rotation to the images according to the specified list of angles. These rotated images are saved in the 'rotation' directory. Finally, the augmented images are displayed in a grid format, showcasing the original images alongside their rotated versions at different angles.

The final adjustments to the rotation angles will determine the orientation of the photographs. Positive angles rotate the image clockwise, while negative angles rotate it counterclockwise. A value of 0 will leave the image unchanged. The magnitude of the angle determines the extent of rotation, with larger angles resulting in more significant rotations.

Translation:

[21]:
```
# Ensure the 'Translation' directory exists in the 'Results' folder
os.makedirs(os.path.join(results_folder, 'Translation'), exist_ok=True)

# Define function to translate image using first principles
def translate_image(image, dx, dy):
    rows, cols, _ = image.shape
    translated_image = np.zeros_like(image)

    for i in range(rows):
        for j in range(cols):
            new_x = j + dx
            new_y = i + dy

            if 0 <= new_x < cols and 0 <= new_y < rows:
                translated_image[new_y, new_x] = image[i, j]

    return translated_image

# Translation offsets to apply
translations = [(-60, -60), (-30, -30), (5, 5), (25, 25), (60, 60)]

# Generate augmented images with translation
for image_file, img in images:
    for dx, dy in translations:
        translated_img = translate_image(img, dx, dy)

        # Save the translated image in the 'Translation' directory within the␣
 ↪'Results' folder
        output_filename = os.path.join(results_folder, 'Translation',␣
 ↪f'translation_{dx}_{dy}_{image_file}')
        plt.imsave(output_filename, translated_img)

# Display the augmented images
fig, axes = plt.subplots(len(translations), len(images), figsize=(20, 20))
```

```
for i, (dx, dy) in enumerate(translations):
    for j, (image_file, _) in enumerate(images):
        ax = axes[i, j]
        ax.imshow(plt.imread(os.path.join(results_folder, 'Translation',␣
 ↪f'translation_{dx}_{dy}_{image_file}')))
        ax.set_title(f"Translation: ({dx}, {dy})")
        ax.axis('off')

plt.show()
```

The code ensures that augmented photographs will be stored in a directory called "translation." Then, it defines the translate_image() function, which translates images based on the specified offsets using first principles of image processing. This function shifts each pixel of the image by the given displacement in the x and y directions. It calculates the new coordinates of each pixel after

translation and assigns the pixel value to the corresponding location in the translated image. The translated image is returned by the function. Following this, a list of translation offsets ranging from (-60, -60) to (60, 60) pixels is established.

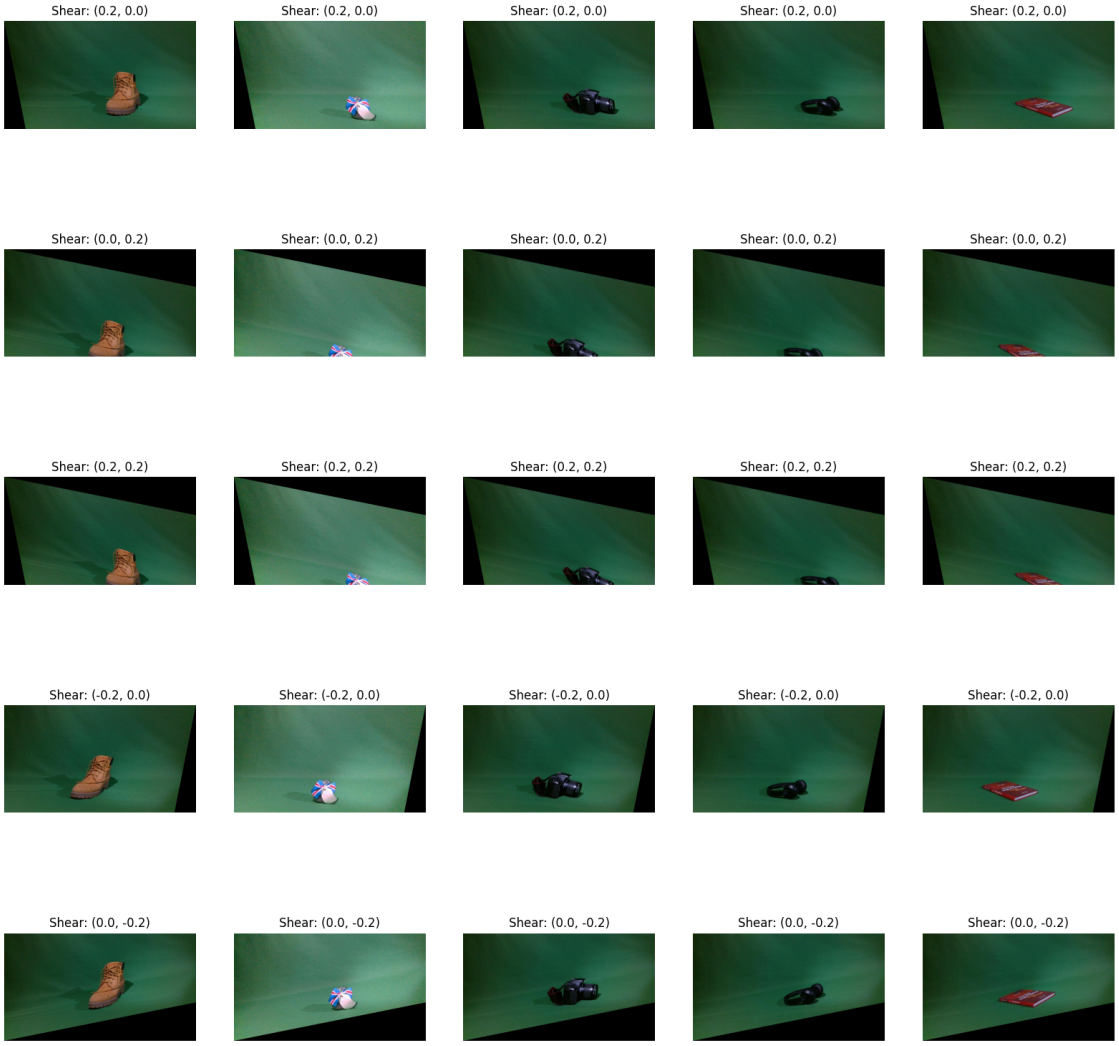
Next, a set of augmented images is generated for each image by applying translation to the images according to the specified list of offsets. These translated images are saved in the 'translation' directory. Finally, the augmented images are displayed in a grid format, showcasing the original images alongside their translated versions with different offsets.

The final adjustments to the translation offsets will determine the displacement of the photographs. Positive values shift the image to the right (in the x-direction) or downwards (in the y-direction), while negative values shift it to the left or upwards, respectively. A value of (0, 0) will leave the image unchanged. The magnitude of the offsets determines the extent of translation, with larger values resulting in more significant shifts.

Shearing:

[22]:
```
# Ensure the 'Shearing' directory exists in the 'Results' folder
os.makedirs(os.path.join(results_folder, 'Shearing'), exist_ok=True)

# Define function to shear image using first principles
def shear_image(image, shear_factor_x, shear_factor_y):
    rows, cols, _ = image.shape
    sheared_image = np.zeros_like(image)

    for i in range(rows):
        for j in range(cols):
            new_x = int(j + shear_factor_x * i)
            new_y = int(i + shear_factor_y * j)

            if 0 <= new_x < cols and 0 <= new_y < rows:
                sheared_image[new_y, new_x] = image[i, j]

    return sheared_image

# Shear factors to apply
shear_factors = [
    (0.2, 0.0),  # Shear only in x-direction
    (0.0, 0.2),  # Shear only in y-direction
    (0.2, 0.2),  # Shear in both directions
    (-0.2, 0.0), # Negative shear in x-direction
    (0.0, -0.2)  # Negative shear in y-direction
]

# Generate augmented images with shearing
for image_file, img in images:
    for shear_factor_x, shear_factor_y in shear_factors:
        sheared_img = shear_image(img, shear_factor_x, shear_factor_y)
```

```
        # Save the sheared image in the 'Shearing' directory within the␣
↪'Results' folder
        output_filename = os.path.join(results_folder, 'Shearing',␣
↪f'shear_{shear_factor_x}_{shear_factor_y}_{image_file}')
        plt.imsave(output_filename, sheared_img)

# Display the augmented images
fig, axes = plt.subplots(len(shear_factors), len(images), figsize=(20, 20))

for i, (shear_factor_x, shear_factor_y) in enumerate(shear_factors):
    for j, (image_file, _) in enumerate(images):
        ax = axes[i, j]
        ax.imshow(plt.imread(os.path.join(results_folder, 'Shearing',␣
↪f'shear_{shear_factor_x}_{shear_factor_y}_{image_file}')))
        ax.set_title(f"Shear: ({shear_factor_x}, {shear_factor_y})")
        ax.axis('off')

plt.show()
```



The code ensures that augmented photographs will be stored in a directory called "shearing." Then, it defines the shear_image() function, which applies shearing to images based on the specified shear factors using first principles of image processing. This function distorts each pixel of the image by shifting it along the x and y axes proportional to its y and x coordinates, respectively. It calculates the new coordinates of each pixel after shearing and assigns the pixel value to the corresponding location in the sheared image. The sheared image is returned by the function. Following this, a list of shear factors ranging from (0.2, 0.0) to (0.0, -0.2) is established.

Next, a set of augmented images is generated for each image by applying shearing to the images according to the specified list of shear factors. These sheared images are saved in the 'shearing' directory. Finally, the augmented images are displayed in a grid format, showcasing the original images alongside their sheared versions with different shear factors.

The final adjustments to the shear factors will determine the distortion of the photographs. Positive shear factors shear the image in the direction of the corresponding axis, while negative shear factors shear it in the opposite direction. A shear factor of (0, 0) will leave the image unchanged. The

magnitude of the shear factors determines the extent of shearing, with larger values resulting in more significant distortions.