01_Img2LUT_CNN_BW_LUT

February 26, 2025

1 Experiment: Learning a 3D LUT with a CNN

In this first experiment, the goal is to prove the general learnability and generation of 3D LUTs with a Convolutional Neural Network (CNN). The CNN receives an RGB image as input and outputs a 3D LUT instead of an image as in the FilmGAN experiment (https://www.researchgate.net/publication/376411929_FilmGAN_towards_realistic_analog_film_photo_generation). The underlying assumption of the experiment is that the generation of a 3D LUT is comparable to the generation of images. To prove the general learnability of 3D LUTs, we attempt to learn a LUT that was used to transform a set of images. The dataset for this experiment consists of paired images, the original images, and the same images after the application of a black-and-white LUT. The image resolution is 256 x 256 pixels with 3 channels (RGB).

2 Imports

[1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
import numpy as np
from torchvision import transforms
from tqdm import tqdm
from PIL import Image
from IPython.display import display
import os
from torchinfo import summary
from pytorch_msssim import ssim, ms_ssim
import torch.nn.functional as F
import matplotlib.pyplot as plt
import torchvision.utils as vutils
from torch.optim.lr_scheduler import ReduceLROnPlateau
```

[2]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

3 Hyperparameters

[3]:
```
batch_size = 64
lut_size = 33
learning_rate = 1e-2
num_epochs = 500
use_interpolation = True
experiment = "bw_w_interpolation"
model_name = "img2lut_bw_w_interpolation.pth.tar"
```

4 Dataset

The Dataset for this Experiment consists of paired images, the original images, and the same images after the application of a black-and-white LUT.

[4]:
```
class PairedImageDataset(Dataset):
    def __init__(self, ungraded_images, graded_images, transform=None):
        self.ungraded_images = ungraded_images
        self.graded_images = graded_images
        self.transform = transform

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

    def __getitem__(self, idx):
        # Returns an ungraded and a graded image
        ungraded = self.ungraded_images[idx]
        graded = self.graded_images[idx]
        if self.transform:
            ungraded = self.transform(ungraded)
            graded = self.transform(graded)
        return ungraded, graded
```

[5]:
```
transform64 = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.ToTensor(),
])
```

[6]:
```
transform = transforms.Compose([
    transforms.ToTensor(),
])
```

The function `load_images_from_directory` gets all images that are within a specific directory, converts them into RGB and can apply a transformation to the images. Additionally the number of images that are processed can be limited with the `num_images` parameter. The function returns a tensor of images. The tensor has the shape (number of images x width x height x channels).

[7]:
```
def load_images_from_directory(directory_path, num_images=100,␣
↪apply_transform=True):
    images = []
    files = os.listdir(directory_path)
    # Ensure only images are processed
    image_files = [f for f in files if f.lower().endswith(('.png', '.jpg', '.
↪jpeg'))]

    for i, image_file in enumerate(image_files[:num_images]):
        image_path = os.path.join(directory_path, image_file)
        img = Image.open(image_path).convert('RGB')  # Convert to RGB in case␣
↪of grayscale
        if apply_transform:
            img = transform64(img)  # Apply the transformations

        else:
            img = transform(img)
        images.append(img)

    images_tensor = torch.stack(images)
    images_tensor = images_tensor.permute(0, 2, 3, 1)
    return images_tensor

ungraded_images = load_images_from_directory('../../analogdb_images_scaled/
↪kodak_ultramax', num_images=400)
graded_images = load_images_from_directory('../../analogdb_images_scaled/
↪kodak_ultramax/BW', num_images=400)

# Verify the shape of the tensors
print(ungraded_images.shape)
print(graded_images.shape)
```

```
torch.Size([317, 256, 256, 3])
torch.Size([317, 256, 256, 3])
```

As we can see above, for this experiment we use 317 images.

[8]:
```
ungraded_images_np = ungraded_images.numpy()
graded_images_np = graded_images.numpy()

dataset = PairedImageDataset(ungraded_images_np, graded_images_np, transform)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

5 Visualization of the Dataset

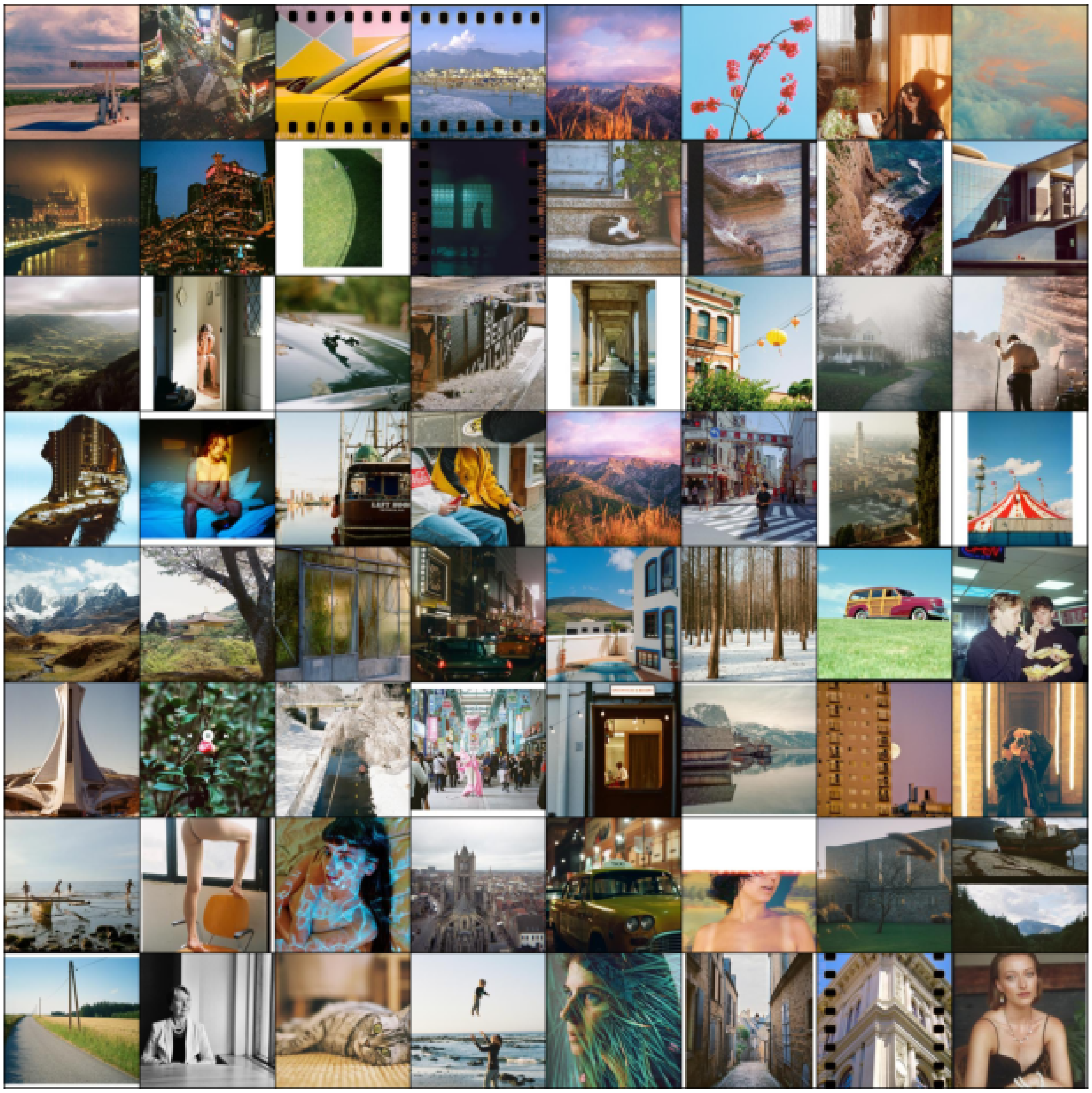
[9]:
```
# Function to plot a set of images in a 8x8 grid
def plot_image_grid(images, title):
    plt.figure(figsize=(8,8))
    plt.axis("off")
    plt.title(title)
    plt.imshow(np.transpose(vutils.make_grid(images.to(device)[:64], padding=2,␣
 ↪normalize=True).cpu(),(1,2,0)))
    plt.show()
```

5.1 Original training Images

The following Image Grid shows the original training images:

[10]:
```
ungraded, graded = next(iter(dataloader))

plot_image_grid(ungraded, "Training Images (Original files)")
```

Training Images (Original files)



5.2 Edited training Images

The following Image Grid shows the edited training images:

[11]:
```
plot_image_grid(graded, "Training Images (After application of BW 3D LUT)")
```

Training Images (After application of BW 3D LUT)

6 ResNet

The model used for this experiment is based on the Residual Network (ResNet) architecture proposed in the paper `Deep Residual Learning for Image Recognition`: https://arxiv.org/pdf/1512.03385

[17]:
```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,␣
↪stride=stride, padding=1)
```

```
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,␣
↪stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(out_channels)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1,␣
↪stride=stride),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        residual = x
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(residual)
        return F.relu(out)
```

Instead of the initial 7x7 convolutional layer, we use a 3x3 filter with a padding of one in the initial convolutional layer that keeps the initial resolution, in this case at 256 x 256. As another measure to adapt the model to lower image resolutions, we use just three residual blocks in comparison to four in the original implementation. The residual blocks are identical to the ResNet implementation with a filter size of 3x3. Just as mentioned in the paper, "when the shortcuts go across feature maps of two sizes, they are performed with a stride of 2". In general, convolutional layers are followed by batch normalization and a ReLU activation function as in the original implementation. After the residual blocks, global average pooling is performed, also following the original implementation. Where the original implementation ends with a 1000-way fully connected layer with a softmax activation function for the task of classifying the 1000 classes in the ImageNet 2012 classification dataset, our model is adopted to generate 3D LUTs. The global average pooling layer is followed by a fully connected layer with 512 dimensions followed by ReLU activation and another fully connected layer that expands the output to the required dimensions for the 3D LUT, which is: LUT_size^3 * channels. The model can be initialized with varying LUT sizes; for an 8 bit LUT the last fully connected layer has 8 * 8 * 8 * 3 = 1536 dimensions.

Source: https://arxiv.org/pdf/1512.03385

[18]:
```
class ResNetLUTGenerator(nn.Module):
    def __init__(self, lut_size=8, trilinear=True, input_channels=3):
        super(ResNetLUTGenerator, self).__init__()
        self.lut_size = lut_size
        self.trilinear = trilinear

        # Initial convolution
        self.initial_conv = nn.Sequential(
            nn.Conv2d(input_channels, 32, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(32),
```

```
            nn.ReLU()
        )

        # ResNet feature extraction
        self.layer1 = self._make_layer(32, 64, stride=2)
        self.layer2 = self._make_layer(64, 128, stride=2)
        self.layer3 = self._make_layer(128, 256, stride=2)

        # Global average pooling
        self.global_pool = nn.AdaptiveAvgPool2d(1)

        # LUT generator
        self.lut_generator = nn.Sequential(
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, lut_size * lut_size * lut_size * 3)
        )

    def _make_layer(self, in_channels, out_channels, stride):
        return ResidualBlock(in_channels, out_channels, stride)


    # Inspiration: https://github.com/HuiZeng/Image-Adaptive-3DLUT/issues/14
    def _trilinear_interpolation(self,luts, images):
        # Scale images to be between -1 and 1
        img = (images - 0.5) * 2.0  # Shape: [4, 3, 256, 256]

        # Add a singleton dimension to represent "channel" for LUT interpolation
        img = img.permute(0, 2, 3, 1).unsqueeze(1)  # Shape: [4, 1, 256, 256, 3]

        # Ensure LUT is in the correct format
        LUT = luts.permute(0, 4, 1, 2, 3)  # Shape: [4, 3, 8, 8, 8]

        # Perform grid sampling for each channel
        result = F.grid_sample(LUT, img, mode='bilinear',␣
↪padding_mode='border', align_corners=True)  # [4, 3, 1, 256, 256]

        # Remove the singleton dimension and permute to the correct format
        result = result.squeeze(2)  # Remove the extra "depth" dimension: [4,␣
↪3, 256, 256]

        return result

    def _simple_approach(self, luts, images):
        lut = luts.view(luts.shape[0], luts.shape[2], luts.shape[2], luts.
↪shape[2], 3)
        image_normalized = (images * luts.shape[2]-1).long()
```

```
        image_normalized = torch.clamp(image_normalized, 0, luts.shape[2]-1)

        r = image_normalized[:, 0, :, :]
        g = image_normalized[:, 1, :, :]
        b = image_normalized[:, 2, :, :]

        transformed = lut[torch.arange(luts.shape[0]).unsqueeze(-1).
↪unsqueeze(-1), r, g, b]
        transformed = transformed.permute(0, 3, 1, 2)

        return transformed


    def forward(self, x):
        # Feature extraction
        x_orig = x  # Store original input
        x = self.initial_conv(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)

        # Global pooling and feature compression
        features = self.global_pool(x).view(x.size(0), -1)

        # Generate LUT
        lut = self.lut_generator(features)
        lut = lut.view(-1, self.lut_size, self.lut_size, self.lut_size, 3)
        lut = torch.sigmoid(lut)

        # Apply LUT to original input
        if self.trilinear:
            transformed = self._trilinear_interpolation(lut, x_orig)
        else:
            transformed = self._simple_approach(lut, x_orig)
        return transformed, lut
```

6.1 Test LUT Generator

[19]:
```
sample_input = torch.rand(4, 3, 256, 256).to(device)
LUTGeneratorCNN=ResNetLUTGenerator().to(device)
transformed, lut = LUTGeneratorCNN(sample_input)
print(lut.shape)
print(transformed.shape)
```

```
torch.Size([4, 8, 8, 8, 3])
torch.Size([4, 3, 256, 256])
```

The models architectural summary for 256 x 256 images and an 8 bit LUT is shown in the summary below, where we can see the initial convolutional layer, followed by the three residual blocks, the global average pooling layer and the fully connected layers for the 3D LUT.

[20]:
```
test_model = ResNetLUTGenerator()
summary(test_model, input_size=(32, 3, 256, 256))
```

[20]:
```
================================================================================
==========
Layer (type:depth-idx)                   Output Shape              Param #
================================================================================
==========
ResNetLUTGenerator                        [32, 3, 256, 256]         --
 Sequential: 1-1                         [32, 32, 256, 256]        --
     Conv2d: 2-1                         [32, 32, 256, 256]        896
     BatchNorm2d: 2-2                    [32, 32, 256, 256]        64
     ReLU: 2-3                           [32, 32, 256, 256]        --
 ResidualBlock: 1-2                      [32, 64, 128, 128]        --
     Conv2d: 2-4                         [32, 64, 128, 128]        18,496
     BatchNorm2d: 2-5                    [32, 64, 128, 128]        128
     Conv2d: 2-6                         [32, 64, 128, 128]        36,928
     BatchNorm2d: 2-7                    [32, 64, 128, 128]        128
     Sequential: 2-8                     [32, 64, 128, 128]        --
         Conv2d: 3-1                     [32, 64, 128, 128]        2,112
         BatchNorm2d: 3-2                [32, 64, 128, 128]        128
 ResidualBlock: 1-3                      [32, 128, 64, 64]         --
     Conv2d: 2-9                         [32, 128, 64, 64]         73,856
     BatchNorm2d: 2-10                   [32, 128, 64, 64]         256
     Conv2d: 2-11                        [32, 128, 64, 64]         147,584
     BatchNorm2d: 2-12                   [32, 128, 64, 64]         256
     Sequential: 2-13                    [32, 128, 64, 64]         --
         Conv2d: 3-3                     [32, 128, 64, 64]         8,320
         BatchNorm2d: 3-4                [32, 128, 64, 64]         256
 ResidualBlock: 1-4                      [32, 256, 32, 32]         --
     Conv2d: 2-14                        [32, 256, 32, 32]         295,168
     BatchNorm2d: 2-15                   [32, 256, 32, 32]         512
     Conv2d: 2-16                        [32, 256, 32, 32]         590,080
     BatchNorm2d: 2-17                   [32, 256, 32, 32]         512
     Sequential: 2-18                    [32, 256, 32, 32]         --
         Conv2d: 3-5                     [32, 256, 32, 32]         33,024
         BatchNorm2d: 3-6                [32, 256, 32, 32]         512
 AdaptiveAvgPool2d: 1-5                  [32, 256, 1, 1]           --
 Sequential: 1-6                         [32, 1536]                --
     Linear: 2-19                        [32, 512]                 131,584
     ReLU: 2-20                          [32, 512]                 --
     Linear: 2-21                        [32, 1536]                787,968
================================================================================
==========
```

```
Total params: 2,128,768
Trainable params: 2,128,768
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 92.28
================================================================================
==========
Input size (MB): 25.17
Forward/backward pass size (MB): 3892.84
Params size (MB): 8.52
Estimated Total Size (MB): 3926.52
================================================================================
==========
```

7 Train Function

[21]:
```python
def display_image(image, width=256):
    tensor_image = image.detach().cpu()
    # Rearrange to HWC format and scale to [0, 255]
    image_np = np.transpose(tensor_image.numpy(), (1, 2, 0))  # Select the␣
↪first image in batch and permute to HWC
    image_np = (image_np * 255).clip(0, 255).astype(np.uint8)

    image = Image.fromarray(image_np)
    display(image)
```

[22]:
```python
def display_images(images, titles=None):
    # Three subplots in one row
    fig, axes = plt.subplots(1, 3, figsize=(4, 2))

    # Convert and display each image
    for idx, (ax, img) in enumerate(zip(axes, [images[0], images[1],␣
↪images[2]])):
        # Convert tensor to numpy array
        tensor_image = img.detach().cpu()
        image_np = np.transpose(tensor_image.numpy(), (1, 2, 0))
        image_np = (image_np * 255).clip(0, 255).astype(np.uint8)

        # Display the image
        ax.imshow(image_np)
        ax.axis('off')

        # Set title if provided
        if titles and idx < len(titles):
            ax.set_title(titles[idx])

    # Adjust layout to prevent overlap
```

```
    plt.tight_layout()
    plt.show()
```

[23]:

```
def l2_regularization(lut):
    # Penalize large deviations in LUT values.
    return torch.mean(lut ** 2)

def smoothness_regularization(lut, lambda_smooth=0.001):
    """
    Calculate smoothness regularization for a batch of 3D LUTs that only␣
 ↪enforces
    local consistency between neighboring values.

    Args:
        lut: Tensor of shape (batch_size, 3, lut_size, lut_size, lut_size)
        lambda_smooth: Smoothness weight factor (should be very small)

    Returns:
        Smoothness loss value
    """
    # Calculate differences with immediate neighbors only
    diff_x = torch.abs(lut[:, :, 1:, :, :] - lut[:, :, :-1, :, :])
    diff_y = torch.abs(lut[:, :, :, 1:, :] - lut[:, :, :, :-1, :])
    diff_z = torch.abs(lut[:, :, :, :, 1:] - lut[:, :, :, :, :-1])

    # Calculate variance in local 2x2x2 neighborhoods
    # This helps ensure smooth transitions without forcing any particular␣
 ↪direction
    local_var_x = torch.var(torch.stack([
        lut[:, :, :-1, :-1, :-1],
        lut[:, :, 1:, :-1, :-1],
    ], dim=-1), dim=-1)

    local_var_y = torch.var(torch.stack([
        lut[:, :, :-1, :-1, :-1],
        lut[:, :, :-1, 1:, :-1],
    ], dim=-1), dim=-1)

    local_var_z = torch.var(torch.stack([
        lut[:, :, :-1, :-1, :-1],
        lut[:, :, :-1, :-1, 1:],
    ], dim=-1), dim=-1)

    # Combine both immediate differences and local variance
    smoothness = (
        diff_x.mean() + diff_y.mean() + diff_z.mean() +  # Immediate neighbor␣
 ↪differences
```

```
        local_var_x.mean() + local_var_y.mean() + local_var_z.mean()  # Local␣
↪variance
    )

    return smoothness * lambda_smooth

def calc_lut_loss(lut):
    #l2_loss = l2_regularization(lut)
    l2_loss = 0
    smoothness_loss = smoothness_regularization(lut)
    total_loss = 0.1 * l2_loss + 0.1 * smoothness_loss
    return total_loss
```

[24]:

```
def ssim_loss(generated, target):
        """
        Structural Similarity Index Measure loss
        Args:
            generated: Generated image tensor (B, C, H, W)
            target: Target image tensor (B, C, H, W)
        """
        return 1 - ssim(generated, target, data_range=255, size_average=True)
```

[25]:

```
def train_model(model, dataloader, optimizer, criterion, num_epochs=10):
    model.train()
    epoch_losses = []

    # Create ReduceLROnPlateau scheduler as in the ResNet paper - divides␣
↪learning rate by 10 when the error plateaus  https://arxiv.org/pdf/1512.03385
    scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.1,␣
↪patience=10)

    for epoch in tqdm(range(num_epochs)):
        running_loss = 0.0
        for ungraded, graded in dataloader:
            ungraded, graded = ungraded.to(device), graded.to(device)

            # Forward pass
            optimizer.zero_grad()
            transformed, _ = model(ungraded)

            loss = criterion(transformed, graded) #+ ssim_loss(transformed,␣
↪graded) #+ calc_lut_loss(lut)

            # Backward pass
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
```

```
        avg_epoch_loss = running_loss / len(dataloader)
        epoch_losses.append(avg_epoch_loss)

        # scheduler step with average loss
        scheduler.step(avg_epoch_loss)

        #print(f"Epoch [{epoch + 1}/{num_epochs}], Loss: {running_loss /␣
↪len(dataloader)}")
        if epoch % 50 == 0:
            display_images([ungraded[0], graded[0], transformed[0]], ["raw",␣
↪"jpg", "model output"])

    return epoch_losses
```

8 Training

[26]:
```
model = ResNetLUTGenerator(lut_size=lut_size, trilinear=use_interpolation).
  ↪to(device)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
criterion = nn.MSELoss()
```

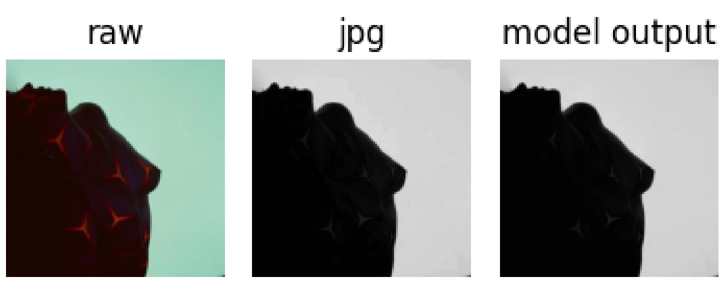
[27]:
```
loss_summary = train_model(model, dataloader, optimizer, criterion, num_epochs)
```

```
  0%|
| 0/500 [00:00<?, ?it/s]
```


```
 10%|
| 50/500 [00:39<06:07,  1.22it/s]
```

20%|
| 100/500 [01:18<05:10, 1.29it/s]

30%|
| 150/500 [01:57<04:28, 1.30it/s]

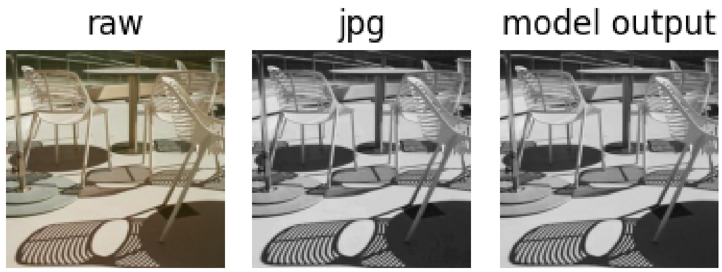


40%|
| 200/500 [02:36<03:51, 1.29it/s]


```
 50%|
| 250/500 [03:15<03:13,  1.29it/s]
```


```
 60%|
| 300/500 [03:54<02:36,  1.28it/s]
```


```
 70%|
| 350/500 [04:33<01:55,  1.29it/s]
```




```
80%|
| 400/500 [05:12<01:18,  1.28it/s]
```

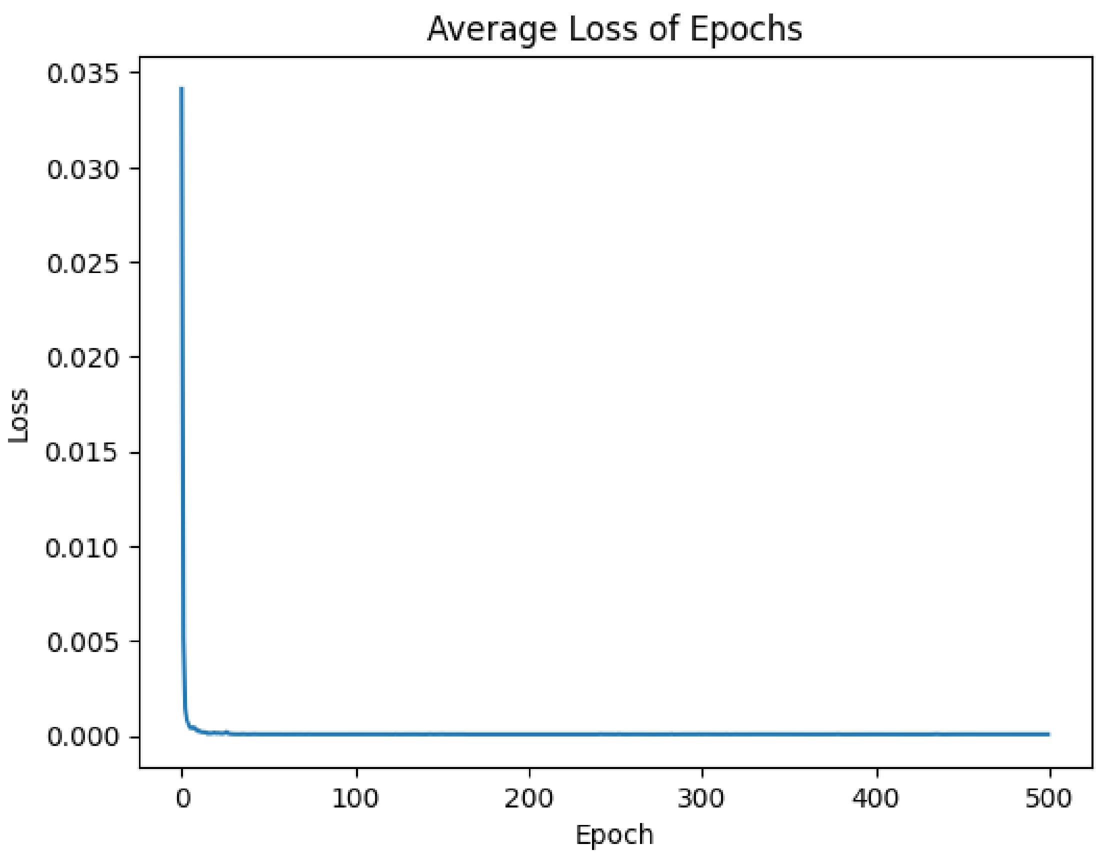

```
90%|
| 450/500 [05:51<00:38,  1.28it/s]
```


```
100%|
  | 500/500 [06:30<00:00,  1.28it/s]
```

[28]:

```
plt.plot(loss_summary)
# Add Title

plt.title("Average Loss of Epochs")
```

```
# Add Axes Labels

plt.xlabel("Epoch")
plt.ylabel("Loss")

# Display

plt.show()
```



[29]:
```
print(loss_summary[-1])
```

```
7.845669606467708e-05
```

8.1 Save Model

[30]:
```
def save_checkpoint(model, optimizer, directory, filename="models/checkpoint.
↪pth.tar"):
    print("=> Saving checkpoint")
    checkpoint = {
        "state_dict": model.state_dict(),
```

```
        "optimizer": optimizer.state_dict(),
    }
    directory = f"models/{directory}"
    # Create the directory if it doesn't exist
    if not os.path.exists(directory):
        os.makedirs(directory)

    torch.save(checkpoint, os.path.join(directory, filename))
```

[31]:
```
save_checkpoint(model, optimizer, directory=experiment, filename=model_name)
```

```
=> Saving checkpoint
```

The evaluation of the models performance on validation images is performed in the Notebook `01_Val_ResNet_BW_Models`.