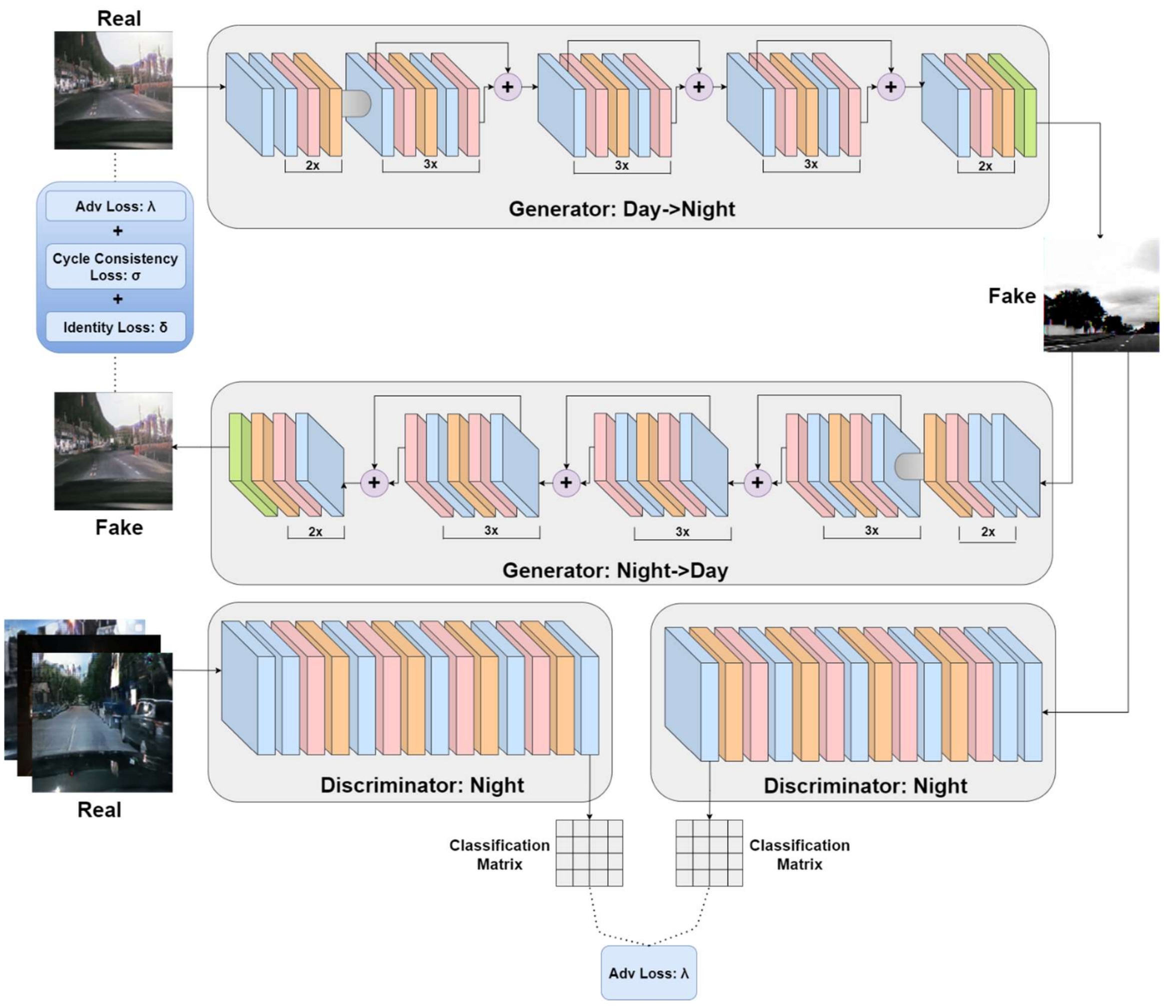
Pervasive AI Developer Contest with AMD

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Shravan Venkatraman – 21BCE1200
Pavan Kumar S – 21BCE1179
Abeshek A – 21BCE5096
Aravintakshan S A – 21BCE1137

B. Tech. Computer Science and Engineering



Augmented Autonomous Vision using Generative Adversarial Networks – Workflow

CONTENTS

Augmented Autonomous Vision using Generative Adversarial Networks

PROJECT OVERVIEW:

PROBLEM DESCRIPTION:

Autonomous Vehicles (AVs) heavily rely on visual data for navigation, obstacle detection, and real-time decision-making. However, varied lighting conditions, such as the transition from day to night, can significantly impact the efficacy of these vision systems. For instance, a system primarily trained using daylight images could face significant challenges when nighttime scenarios are considered. This results in safety risks and diminished operational efficiency.

Therefore, to mitigate this issue, our project aims to leverage the capabilities of Cycle Generative Adversarial Networks (Cycle GANs). These networks are specifically designed for image-to-image translation tasks, offering a unique advantage over traditional Generative Adversarial Networks (GANs). Unlike conventional GANs that require paired images from both domains (such as matching day and night images of the same scene), Cycle GANs can effectively learn from unpaired data. This is helpful in cases where paired datasets are difficult to obtain. By utilizing Cycle GANs, we aim to enhance the adaptability and reliability of AV vision systems across different lighting conditions, ultimately improving safety and performance.

PROJECT DESCRIPTION:

A Cycle GAN is designed for image-to-image translation tasks which doesn't require any paired examples. The architecture of a Cycle-GAN involves the following core components:

- Generators
- Discriminators

Generators:

In Cycle GANs generators are responsible for transforming images from one domain to another (in this case, daytime images to nighttime images and vice versa). The generator's design plays a crucial role in translating the quality of translated images. In our case the generators are used for day-night translation and night-day translation respectively. The day-night translator is considered as generator G and night-day translator is considered as generator F. A generator's architecture is composed of the following components:

- Feature Map Block
- Contracting Block
- Residual Block
- Expanding Block

Thus a generator is developed as a series of 2 contracting blocks, 9 residual blocks, and 2 expanding blocks to transform an input image into an image from the other class, with an upfeature layer at the start and a downfeature layer at the end.

Discriminators:

The discriminator in a Cycle GAN is responsible for distinguishing between real images from the target domain and fake images generated by the generator. Its design is crucial for training the generator to produce realistic images. The discriminator is composed of Feature Map Block and Contracting Blocks which is structured like the contracting path of the U-Net architecture.

Block Architecture:

Feature Map Block:

The Feature Map Block represents a layer within the Generator that is essential for both upsampling and downsampling tasks. This block is primarily designed to adjust the number of channels in the feature maps produced by previous layers to the desired count for subsequent processing. By transforming the feature maps to have the correct number of output channels, the `FeatureMapBlock` ensures that the data passed through the network maintains the appropriate dimensions and consistency required for further operations.

In terms of functionality, the `FeatureMapBlock` takes the input feature maps and processes them to produce an output with a specified number of channels. This transformation is achieved through convolutional operations, which modify the depth of the feature maps while preserving or altering their spatial dimensions depending on the specific task (upsampling or downsampling). The block is versatile and can be used at various stages of the Generator to prepare the feature maps for the next layer, ensuring seamless data flow and compatibility throughout the network.

Expanding Block:

The Expanding Block class is designed for the upsampling process in the network. Its primary function is to increase the spatial dimensions of the input feature maps, which is achieved through a convolutional transpose operation, also known as a deconvolution. This process is essential for generating higher-resolution images from lower-resolution ones, a crucial step in image synthesis tasks. Additionally, the Expanding Block can optionally include an instance normalization step, which helps standardize the feature maps, thus stabilizing and accelerating the training process.

Functionally, the Expanding Block performs the convolutional transpose operation to scale up the size of the feature maps. This involves reversing the downsampling process, effectively increasing the width and height of the input data. The optional instance normalization further processes these upsampled feature maps by normalizing them, which helps maintain the stability of the model during training and improves convergence rates. This combination of upsampling and normalization ensures that the feature maps are not only larger but also properly scaled and standardized for subsequent layers in the network.

Contracting Block:

The Contracting Block class is intended for the downsampling process within the network. It achieves this by performing a convolution operation followed by a max pooling operation, which extracts essential features from the input and reduces the spatial dimensions of the feature maps. This process is crucial for compressing the information into a more manageable form, making it easier for the network to process and analyze complex patterns. Like the Expanding Block, the Contracting Block can also optionally apply instance normalization to the feature maps.

In terms of operation, the Contracting Block first uses a convolution operation to extract features from the input data, capturing important patterns and details. Following this, a max pooling operation is applied to reduce the spatial dimensions of the feature maps, effectively downsampling the data. The optional instance normalization step can then be used to standardize the feature maps, improving the training process by ensuring consistency and stability. This combination of convolution, pooling, and normalization allows the Contracting Block to efficiently downsample the input while preserving crucial information for further processing in the network.

Residual Block:

The Residual Block class is designed to implement a residual connection within the network, a feature that helps in mitigating the vanishing gradient problem and allows for the training of deeper networks. It performs two convolution operations followed by instance normalization. The key aspect of this block is the addition of the input feature maps to the output of these operations, creating a residual connection. This structure ensures that the original information is preserved and passed through the network, facilitating better gradient flow and enabling the network to learn more complex patterns.

Functionally, the Residual Block starts with two consecutive convolution operations, each followed by instance normalization to process the input feature maps. These operations transform the input data, extracting and refining features. After these transformations, the original input feature maps are added to the processed output, creating a residual connection. This addition helps preserve the original input information, ensuring that it is not lost through the transformations. The resulting output is a combination of the original and processed data, which enhances the network's ability to learn and maintain stability during training. This residual connection is critical for allowing deeper networks to be trained effectively, as it helps prevent the degradation of important features and ensures a more robust learning process.

Working:

The process starts with taking an input image x from the source domain X. The generator G is then utilized to transform this input image into a translated image y in the target domain Y. This translated image is subsequently processed by another generator F, generating a reconstructed version of the original image x.

These generators, G and F, are aided by discriminators D_Y and D_X, respectively. The discriminators are responsible for providing critical feedback to the generators by evaluating the realism of the generated images. Specifically, D_Y evaluates the realism of the images in domain Y, and D_X evaluates the realism of the images in domain X.

The generators receive feedback from their corresponding discriminators and learn to generate images that are indistinguishable from real images in domains Y and X. This adversarial training helps each generator refine its output, making the translated images increasingly realistic over time.

Through this cycle-consistent adversarial training, the model aims to achieve realistic image translation and reconstruction, ensuring that the translated images maintain the original image's structure and features.

Loss Functions:

The total loss in the Cycle-GAN is the combination of cycle consistency loss, adversarial loss and identity loss which ensures that the generators produce high-quality, realistic images while maintaining the image translation cycle's consistency and at the same time preserves key attributes of the input images.

Cycle Consistency Loss (σ):

Cycle Consistency Loss function acts as a metric which ensures that the translation of an image from one domain to another and back results in the generation of original image. This function is crucial since this task involves utilization of unpaired image-to-image translation tasks. It enforces the idea that if you start with an image from domain X, translate it to domain Y, and then translate it back to domain X, you should end up with the original image. This can be expressed mathematically as follows:

$$x \to G(x) \to F\big(G(x)\big) \approx x$$

$$y \to G(y) \to F\big(G(y)\big) \approx y$$

$$L_{cycle}(G,F) = \mathbb{E}_{x\sim p_{data}(x)}\left[\left|\left|F\big(G(x)\big) - x\right|\right|_1\right] + \mathbb{E}_{y\sim p_{data}(y)}\left[\left|\left|G\big(F(y)\big) - y\right|\right|_1\right]$$

Where, G is the Generator which translates from domain X to Y and F is the generator that translates from domain Y to X. The loss measures the closeness of F(G(x)) with x and G(F(y)) with y.

Adversarial Loss (λ):

Adversarial Loss acts as the measure which ensures that the images generated are indistinguishable from real images by utilizing a discriminator network. Each domain has its own discriminator, which learns to distinguish real images from generated images. This can be expressed mathematically as:

For generator G (translates images from domain X to domain Y, and discriminator D_Y for domain Y):

$$L_{GAN}(G, D_Y, X, Y) = \mathbb{E}_{y\sim p_{data}(y)}[logD_Y(y)] + \mathbb{E}_{x\sim p_{data}(x)}\left[\log\Big(1 - D_X\big(G(x)\big)\Big)\right]$$

For generator F (translates images from domain Y to domain X, and discriminator D_X for domain X):

$$L_{GAN}(F, D_X, X, Y) = \mathbb{E}_{x\sim p_{data}(x)}[logD_X(x)] + \mathbb{E}_{y\sim p_{data}(y)}\left[\log\left(1 - D_Y\big(F(x)\big)\right)\right]$$

This loss function encourages the generators G and F to produce realistic images that can fool the discriminators D_Y and D_X.

Identity Loss (δ):

Identity Loss helps in preserving the colour composition between the input images and the generated images. This is useful since the domains X and Y are similar and the generators produce images that are not only realistic but also retain the input's attributes.

This can be expressed mathematically as:

For generator G and input y for domain Y:

$$L_{identity}(G, Y) = \mathbb{E}_{y\sim p_{data}(y)}\left[||G(y) - y||_1\right]$$

For generator F and input x for domain X:

$$L_{identity}(F, X) = \mathbb{E}_{x\sim p_{data}(x)}\left[||F(x) - x||_1\right]$$

Bill of Materials (BOM):

1. Hardware Components

- AMD Radeon Instinct GPUs
    - Model: AMD Radeon Instinct
    - Purpose: Accelerates the training of the Cycle-GAN model by providing high computational power necessary for processing large image datasets and optimizing model parameters.
    - Impact: Improved realism and efficiency in training, as well as enabling real-time inference.
- AMD EPYC 7003 Series Processors
    - Model: AMD EPYC 7763
    - Purpose: Supports data preprocessing tasks and GPU computation with its high core count and exceptional memory bandwidth.
    - Impact: Efficient management of data and support for GPU-based training.
- Operating System
    - Version: Linux 5.15.133
    - Purpose: Provides a stable and flexible environment for running the training and development processes.
    - Impact: Ensures compatibility and performance for software and hardware components.

2. Software Components

- PyTorch
    - Purpose: Framework for implementing the Cycle-GAN model, managing the training process, and performing image translation tasks.
    - Impact: Facilitates efficient model development and experimentation.
- Python
    - Purpose: Programming language used for implementing various components of the solution.
    - Impact: Provides the necessary scripting environment for the development and execution of the Cycle-GAN model.
- Jupyter Notebook
    - Purpose: Interactive development and experimentation environment for coding, visualization, and debugging.
    - Impact: Enhances the workflow of model development and testing.
- Git
    - Purpose: Version control system for managing the codebase and collaborating with team members.
    - Impact: Ensures code integrity and facilitates collaborative development.

3. System Configuration

- CPU Specifications
    - Model: AMD EPYC 7763
    - Architecture: x86_64
    - Core(s) per Socket: 64
    - Socket(s): 2

    - Thread(s) per Core: 1
    - Total CPUs: 128
- GPU Specifications
    - Model: AMD Radeon Instinct
    - Number of GPUs: 1

INSTRUCTIONS:

Instructions for Day-Night CycleGAN

This script implements a CycleGAN model for translating images between day and night domains. Follow these instructions to set up and run the CycleGAN model:

Prerequisites

1. Python Libraries: Ensure you have the following libraries installed:

- PyTorch
- torchvision
- numpy
- tqdm
- (Optional) any other dependencies for loading and processing images.

2. Data Preparation:

- Prepare two datasets: one set containing images from the day domain and the other from the night domain.
- The datasets should be organized in a format compatible with the DataLoader. Each dataset should be a directory containing images.

Setup

1. Directory Structure:

- Place your day images in a directory (e.g., `data/day`).
- Place your night images in a separate directory (e.g., `data/night`).

2. Parameters:

- batch_size: Set the batch size for training.
- target_shape: Define the target shape for resizing images.
- n_epochs: Specify the number of training epochs.
- lambda_identity: Weight for the identity loss.
- lambda_cycle: Weight for the cycle consistency loss.
- device: Set the computing device (e.g., 'cuda' for GPU or 'cpu').

Running the Code

1. Loading the Data:

The DataLoader will automatically load images from the specified directories. Ensure that the `dataset` variable is correctly set up to point to your image directories.

2. Training:

Call the `train()` function to start the training process. This function handles the training loop, updates the generators and discriminators, and saves model checkpoints periodically.

- Optionally, set `save_model=True` in the `train()` function to save the trained models.

3. Monitoring:

The training process will print out the generator and discriminator losses at regular intervals defined by `display_step`.

4. Model Checkpoints:

Model checkpoints are saved during training to allow you to resume training or use the model for inference later.

Example

To run the training, simply execute the script. Ensure you have configured the parameters and data directories as needed.

python

Example usage

train(save_model=True)

PROGRAM USED WITH COMMENTS:

```
import torch
from torch import nn
from tqdm.auto import tqdm
from torchvision import transforms
from torchvision.utils import make_grid
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import glob
from torch.utils.data import Dataset
from PIL import Image

# Set random seed for reproducibility
torch.manual_seed(0)

def show_tensor_images(image_tensor, num_images=25, size=(1, 28, 28)):
    '''
    Visualize images in a uniform grid.

    Parameters:
        image_tensor (torch.Tensor): Tensor containing images to visualize.
        num_images (int): Number of images to display.
        size (tuple): Size of each image in the tensor.
    '''
    image_tensor = (image_tensor + 1) / 2
    image_unflat = image_tensor.detach().cpu().view(-1, *size)
    image_grid = make_grid(image_unflat[:num_images], nrow=5)
    plt.imshow(image_grid.permute(1, 2, 0).squeeze())
    plt.show()
```

```
class ImageDataset(Dataset):
    '''
    Custom Dataset class for loading paired images.

    Parameters:
        root (str): Root directory containing 'day' and 'night' folders with images.
        transform (callable, optional): Transformation to be applied on images.
    '''
    def __init__(self, root, transform=None):
        self.transform = transform
        self.files_A = sorted(glob.glob(root + '/day/*.*'))
        self.files_B = sorted(glob.glob(root + '/night/*.*'))
        if len(self.files_A) > len(self.files_B):
            self.files_A, self.files_B = self.files_B, self.files_A
        self.new_perm()
        assert len(self.files_A) > 0, "Make sure you downloaded the images!"

    def new_perm(self):
        self.randperm = torch.randperm(len(self.files_B))[:len(self.files_A)]

    def __getitem__(self, index):
        item_A = self.transform(Image.open(self.files_A[index % len(self.files_A)]))
        item_B = self.transform(Image.open(self.files_B[self.randperm[index]]))
        if item_A.shape[0] != 3:
            item_A = item_A.repeat(3, 1, 1)
        if item_B.shape[0] != 3:
            item_B = item_B.repeat(3, 1, 1)
        if index == len(self) - 1:
            self.new_perm()
        return (item_A - 0.5) * 2, (item_B - 0.5) * 2

    def __len__(self):
        return min(len(self.files_A), len(self.files_B))

class ResidualBlock(nn.Module):
    '''
    Residual Block for applying two convolutions and instance normalization.

    Parameters:
        input_channels (int): Number of input channels.
    '''
    def __init__(self, input_channels):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(input_channels, input_channels, kernel_size=3, padding=1, 
padding_mode='reflect')
        self.conv2 = nn.Conv2d(input_channels, input_channels, kernel_size=3, padding=1, 
padding_mode='reflect')
        self.instancenorm = nn.InstanceNorm2d(input_channels)
        self.activation = nn.ReLU()

    def forward(self, x):
        '''
        Forward pass of ResidualBlock.

        Parameters:
            x (torch.Tensor): Input tensor of shape (batch_size, channels, height, width).

        Returns:
            torch.Tensor: Output tensor with residual connections.
        '''
        original_x = x.clone()
        x = self.conv1(x)
        x = self.instancenorm(x)
        x = self.activation(x)
        x = self.conv2(x)
        x = self.instancenorm(x)
        return original_x + x


class ContractingBlock(nn.Module):
    '''
    Contracting Block for downsampling with convolution and optional instance
normalization.


    Parameters:
        input_channels (int): Number of input channels.
        use_bn (bool): Whether to use instance normalization.
        kernel_size (int): Size of convolution kernel.
        activation (str): Activation function to use ('relu' or 'lrelu').
    '''
    def __init__(self, input_channels, use_bn=True, kernel_size=3, activation='relu'):
        super(ContractingBlock, self).__init__()
        self.conv1 = nn.Conv2d(input_channels, input_channels * 2, kernel_size=kernel_size,
padding=1, stride=2, padding_mode='reflect')
        self.activation = nn.ReLU() if activation == 'relu' else nn.LeakyReLU(0.2)
        if use_bn:
            self.instancenorm = nn.InstanceNorm2d(input_channels * 2)
        self.use_bn = use_bn


    def forward(self, x):
        '''
        Forward pass of ContractingBlock.
        Parameters:
            x (torch.Tensor): Input tensor of shape (batch_size, channels, height, width).

        Returns:
            torch.Tensor: Output tensor after downsampling.
        '''
        x = self.conv1(x)
        if self.use_bn:
            x = self.instancenorm(x)
        x = self.activation(x)
        return x

class ExpandingBlock(nn.Module):
    '''
    Expanding Block for upsampling with convolution transpose and optional instance
normalization.

    Parameters:
        input_channels (int): Number of input channels.
        use_bn (bool): Whether to use instance normalization.
    '''
    def __init__(self, input_channels, use_bn=True):
        super(ExpandingBlock, self).__init__()
        self.conv1 = nn.ConvTranspose2d(input_channels, input_channels // 2, kernel_size=3,
stride=2, padding=1, output_padding=1)
        if use_bn:
            self.instancenorm = nn.InstanceNorm2d(input_channels // 2)
        self.use_bn = use_bn
        self.activation = nn.ReLU()
    def forward(self, x):
        '''
        Forward pass of ExpandingBlock.

        Parameters:
            x (torch.Tensor): Input tensor of shape (batch_size, channels, height, width).

        Returns:
            torch.Tensor: Output tensor after upsampling.
        '''
        x = self.conv1(x)
        if self.use_bn:
            x = self.instancenorm(x)
        x = self.activation(x)
        return x

class FeatureMapBlock(nn.Module):
    '''
    Final layer of a Generator to map output to desired number of channels.

    Parameters:
        input_channels (int): Number of input channels.
        output_channels (int): Number of output channels.
    '''
    def __init__(self, input_channels, output_channels):
        super(FeatureMapBlock, self).__init__()
        self.conv = nn.Conv2d(input_channels, output_channels, kernel_size=7, padding=3,
padding_mode='reflect')

    def forward(self, x):
        '''
        Forward pass of FeatureMapBlock.


        Parameters:
            x (torch.Tensor): Input tensor of shape (batch_size, channels, height, width).


        Returns:
            torch.Tensor: Output tensor mapped to the desired number of channels.
        '''
        x = self.conv(x)
        return x


class Generator(nn.Module):
    '''
    Generator Class implementing U-Net with residual blocks.


    Parameters:
        input_channels (int): Number of input channels.
        output_channels (int): Number of output channels.
        hidden_channels (int): Number of hidden channels.
    '''
    def __init__(self, input_channels, output_channels, hidden_channels=64):
        super(Generator, self).__init__()
        self.upfeature = FeatureMapBlock(input_channels, hidden_channels)
        self.contract1 = ContractingBlock(hidden_channels)
        self.contract2 = ContractingBlock(hidden_channels * 2)
        res_mult = 4
        self.res_blocks = nn.Sequential(
            *[ResidualBlock(hidden_channels * res_mult) for _ in range(9)]
        )
        self.expand2 = ExpandingBlock(hidden_channels * 4)
        self.expand3 = ExpandingBlock(hidden_channels * 2)
        self.downfeature = FeatureMapBlock(hidden_channels, output_channels)
        self.tanh = torch.nn.Tanh()

    def forward(self, x):
        '''
        Forward pass of Generator.

        Parameters:
            x (torch.Tensor): Input tensor of shape (batch_size, channels, height, width).

        Returns:
            torch.Tensor: Output tensor transformed to target domain.
        '''
        x0 = self.upfeature(x)
        x1 = self.contract1(x0)
        x2 = self.contract2(x1)
        x3 = self.res_blocks(x2)
        x12 = self.expand2(x3)
        x13 = self.expand3(x12)
        xn = self.downfeature(x13)
        return self.tanh(xn)

class Discriminator(nn.Module):
    '''
    Discriminator Class for classifying real/fake images.

    Parameters:
        input_channels (int): Number of input channels.
        hidden_channels (int): Number of initial convolutional filters.
    '''
    def __init__(self, input_channels, hidden_channels=64):
        super(Discriminator, self).__init__()
        self.upfeature = FeatureMapBlock(input_channels, hidden_channels)
        self.contract1 = ContractingBlock(hidden_channels, use_bn=False, kernel_size=4,
activation='lrelu')
        self.contract2 = ContractingBlock(hidden_channels * 2, kernel_size=4,
activation='lrelu')
        self.contract3 = ContractingBlock(hidden_channels * 4, kernel_size=4,
activation='lrelu')
        self.conv4 = nn.Conv2d(hidden_channels * 8, 1, kernel_size=4)
        self.activation = nn.Sigmoid()

    def forward(self, x):
        '''
        Forward pass of Discriminator.

        Parameters:
            x (torch.Tensor): Input tensor of shape (batch_size, channels, height, width).

        Returns:
            torch.Tensor: Output tensor indicating real/fake classification.
        '''
        x = self.upfeature(x)
        x = self.contract1(x)
        x = self.contract2(x)
        x = self.contract3(x)
        x = self.conv4(x)
        return self.activation(x)

class TrainingConfig:
    '''
    Configuration class for training the GAN model.


    Parameters:
        lr (float): Learning rate.
        beta1 (float): Beta1 parameter for Adam optimizer.
        beta2 (float): Beta2 parameter for Adam optimizer.
        lambda_cycle (float): Weight for cycle consistency loss.
        lambda_identity (float): Weight for identity loss.
    '''
    def __init__(self, lr=0.0002, beta1=0.5, beta2=0.999, lambda_cycle=10.0,
lambda_identity=0.5):
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.lambda_cycle = lambda_cycle
        self.lambda_identity = lambda_identity
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


def create_optimizers(generator, discriminator, config):
    '''
    Create Adam optimizers for the generator and discriminator.


    Parameters:
        generator (nn.Module): The Generator network.
        discriminator (nn.Module): The Discriminator network.
        config (TrainingConfig): Configuration for training.


    Returns:
        Tuple[torch.optim.Adam, torch.optim.Adam]: Optimizers for generator and
discriminator.
```

```
    '''
    optim_G = torch.optim.Adam(generator.parameters(), lr=config.lr, betas=(config.beta1,
config.beta2))
    optim_D = torch.optim.Adam(discriminator.parameters(), lr=config.lr, betas=(config.beta1,
config.beta2))
    return optim_G, optim_D


def create_losses(config):
    '''
    Create loss functions for GAN training.

    Parameters:
        config (TrainingConfig): Configuration for training.

    Returns:
        Tuple[nn.Module, nn.Module]: Loss functions for generator and discriminator.
    '''
    criterion_GAN = nn.MSELoss()
    criterion_cycle = nn.L1Loss()
    criterion_identity = nn.L1Loss()
    return criterion_GAN, criterion_cycle, criterion_identity

# Define model parameters
input_channels = 3
output_channels = 3
hidden_channels = 64
config = TrainingConfig()

# Initialize models
generator = Generator(input_channels, output_channels, hidden_channels).to(config.device)
discriminator = Discriminator(input_channels, hidden_channels).to(config.device)
```

```
# Create optimizers and loss functions
optim_G, optim_D = create_optimizers(generator, discriminator, config)
criterion_GAN, criterion_cycle, criterion_identity = create_losses(config)

# Display sample images
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(256),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

dataset = ImageDataset('data', transform=transform)
dataloader = DataLoader(dataset, batch_size=4, shuffle=True)
real_A, real_B = next(iter(dataloader))
show_tensor_images(real_A, num_images=10, size=(3, 256, 256))
show_tensor_images(real_B, num_images=10, size=(3, 256, 256))

def weights_init(m):
    if isinstance(m, nn.Conv2d) or isinstance(m, nn.ConvTranspose2d):
        torch.nn.init.normal_(m.weight, 0.0, 0.02)
    if isinstance(m, nn.BatchNorm2d):
        torch.nn.init.normal_(m.weight, 0.0, 0.02)
        torch.nn.init.constant_(m.bias, 0)

# Feel free to change pretrained to False if you're training the model from scratch
pretrained = False
if pretrained:
    pre_dict = torch.load('cycleGAN_100000.pth')
    gen_AB.load_state_dict(pre_dict['gen_AB'])
    gen_BA.load_state_dict(pre_dict['gen_BA'])
    gen_opt.load_state_dict(pre_dict['gen_opt'])
    disc_A.load_state_dict(pre_dict['disc_A'])
    disc_A_opt.load_state_dict(pre_dict['disc_A_opt'])
    disc_B.load_state_dict(pre_dict['disc_B'])
    disc_B_opt.load_state_dict(pre_dict['disc_B_opt'])
else:
    gen_AB = gen_AB.apply(weights_init)
    gen_BA = gen_BA.apply(weights_init)
    disc_A = disc_A.apply(weights_init)
    disc_B = disc_B.apply(weights_init)
```

""""

## CycleGAN for Day-Night Image Translation

This implementation of CycleGAN focuses on translating images between day and night domains. The generator models `gen_AB` and `gen_BA` perform the translation from day to night and from night to day, respectively. Discriminators `disc_A` and `disc_B` evaluate the authenticity of the generated images for the day and night domains.

### Functions Overview:

1. **get_gen_adversarial_loss**: Computes the adversarial loss for a generator.

2. **get_identity_loss**: Calculates the identity loss to preserve color and structure.

3. **get_cycle_consistency_loss**: Measures the cycle consistency to ensure that translating an image to the target domain and back returns the original image.

4. **get_gen_loss**: Combines adversarial, identity, and cycle consistency losses to compute the total generator loss.

5. **train**: Trains the CycleGAN model by alternating updates between generators and discriminators.

""""

```
def get_gen_adversarial_loss(real_X, disc_X, gen_XY, adv_criterion):
    '''
    Return the adversarial loss of the generator given inputs (and the generated images for
testing purposes).
    Parameters:
        real_X: the real images from the day domain
        disc_X: the discriminator for the day domain; takes images and returns real/fake class
day prediction matrices
        gen_XY: the generator for day to night; takes day images and returns night images
        adv_criterion: the adversarial loss function; takes the discriminator predictions and the
target labels and returns an adversarial loss (which you aim to minimize)
    '''
    fake_Y = gen_XY(real_X)
    fake_Y_pred = disc_X(fake_Y)
    adv_loss = adv_criterion(fake_Y_pred, torch.ones_like(fake_Y_pred))
    return adv_loss, fake_Y


def get_identity_loss(real_X, gen_YX, identity_criterion):
    '''
    Return the identity loss of the generator given inputs (and the generated images for testing
purposes).
    Parameters:
        real_X: the real images from the day domain
        gen_YX: the generator for night to day; takes night images and returns day images
        identity_criterion: the identity loss function; takes the real images from the day domain
and those images put through the night-to-day generator and returns the identity loss (which
you aim to minimize)
    '''
    identity_X = gen_YX(real_X)
    identity_loss = identity_criterion(identity_X, real_X)
    return identity_loss, identity_X
```

```
def get_cycle_consistency_loss(real_X, fake_Y, gen_YX, cycle_criterion):
    '''
    Return the cycle consistency loss of the generator given inputs (and the generated images
for testing purposes).
    Parameters:
        real_X: the real images from the day domain
        fake_Y: the fake images from the night domain, generated from real_X
        gen_YX: the generator for night to day; takes night images and returns the images
transformed to the day domain
        cycle_criterion: the cycle loss function; takes the real images from the day domain and
those images put through the night-to-day generator and returns the cycle loss (which you
aim to minimize)
    '''
    cycle_X = gen_YX(fake_Y)
    cycle_loss = cycle_criterion(cycle_X, real_X)
    return cycle_loss, cycle_X


def get_gen_loss(real_A, real_B, gen_AB, gen_BA, disc_A, disc_B, adv_criterion,
identity_criterion, cycle_criterion, lambda_identity=0.1, lambda_cycle=10):
    '''
    Return the generator loss of the generator given inputs (and the generated images for
testing purposes).
    Parameters:
        real_A: the real images from the day domain
        real_B: the real images from the night domain
        gen_AB: the generator for day to night; takes day images and returns night images
        gen_BA: the generator for night to day; takes night images and returns day images
        disc_A: the discriminator for the day domain; takes images and returns real/fake class
day prediction matrices
        disc_B: the discriminator for the night domain; takes images and returns real/fake class
night prediction matrices
        adv_criterion: the adversarial loss function; takes the discriminator predictions and the
target labels and returns an adversarial loss (which you aim to minimize)
```

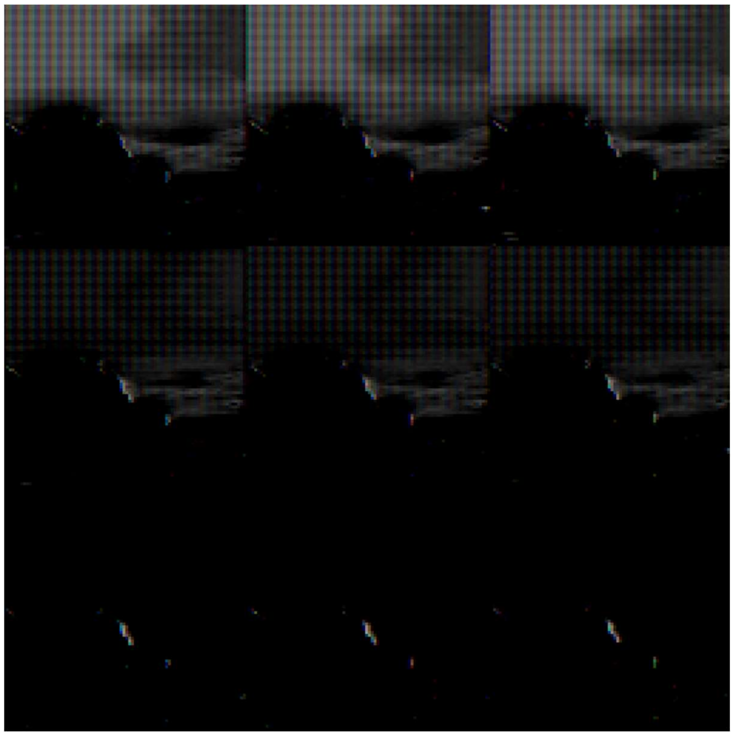
```
        identity_criterion: the identity loss function; takes the real images from the day domain and those images put through the night-to-day generator and returns the identity loss (which you aim to minimize)
        cycle_criterion: the cycle loss function; takes the real images from the day domain and those images put through the night-to-day generator and returns the cycle loss (which you aim to minimize)
        lambda_identity: weight for identity loss
        lambda_cycle: weight for cycle loss
    '''
    # Calculate adversarial loss
    adv_loss_BA, fake_A = get_gen_adversarial_loss(real_B, disc_A, gen_BA, adv_criterion)
    adv_loss_AB, fake_B = get_gen_adversarial_loss(real_A, disc_B, gen_AB, adv_criterion)
    gen_adversarial_loss = adv_loss_BA + adv_loss_AB

    # Calculate identity loss
    identity_loss_A, identity_A = get_identity_loss(real_A, gen_BA, identity_criterion)
    identity_loss_B, identity_B = get_identity_loss(real_B, gen_AB, identity_criterion)
    gen_identity_loss = identity_loss_A + identity_loss_B

    # Calculate cycle consistency loss
    cycle_loss_BA, cycle_A = get_cycle_consistency_loss(real_A, fake_B, gen_BA, cycle_criterion)
    cycle_loss_AB, cycle_B = get_cycle_consistency_loss(real_B, fake_A, gen_AB, cycle_criterion)
    gen_cycle_loss = cycle_loss_BA + cycle_loss_AB

    # Combine losses
    gen_loss = lambda_identity * gen_identity_loss + lambda_cycle * gen_cycle_loss + gen_adversarial_loss
    return gen_loss, fake_A, fake_B

def train(save_model=True):
    mean_generator_loss = 0
```

```
mean_discriminator_loss = 0
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
cur_step = 0

for epoch in range(n_epochs):
    for real_A, real_B in tqdm(dataloader):
        # Resize images to the target shape
        real_A = nn.functional.interpolate(real_A, size=target_shape)
        real_B = nn.functional.interpolate(real_B, size=target_shape)

        # Move images to device
        cur_batch_size = len(real_A)
        real_A = real_A.to(device)
        real_B = real_B.to(device)

        ### Update discriminator for the day domain ###
        disc_A_opt.zero_grad()
        with torch.no_grad():
            fake_A = gen_BA(real_B)
        disc_A_loss = get_disc_loss(real_A, fake_A, disc_A, adv_criterion)
        disc_A_loss.backward(retain_graph=True)
        disc_A_opt.step()

        ### Update discriminator for the night domain ###
        disc_B_opt.zero_grad()
        with torch.no_grad():
            fake_B = gen_AB(real_A)
        disc_B_loss = get_disc_loss(real_B, fake_B, disc_B, adv_criterion)
        disc_B_loss.backward(retain_graph=True)
        disc_B_opt.step()
        ### Update generator ###
        gen_opt.zero_grad()
        gen_loss, fake_A, fake_B = get_gen_loss(
            real_A, real_B, gen_AB, gen_BA, disc_A, disc_B, adv_criterion,
identity_criterion, cycle_criterion
        )
        gen_loss.backward()
        gen_opt.step()

        # Update mean losses and display results
        mean_discriminator_loss += disc_A_loss.item() / display_step
        mean_generator_loss += gen_loss.item() / display_step

        if cur_step % display_step == 0:
            print(f"Epoch {epoch}: Step {cur_step}: Generator (U-Net) loss:
{mean_generator_loss}, Discriminator loss: {mean_discriminator_loss}")
            mean_generator_loss = 0
            mean_discriminator_loss = 0

            # Save model checkpoints
            if save_model:
                torch.save({
                    'gen_AB': gen_AB.state_dict(),
                    'gen_BA': gen_BA.state_dict(),
                    'gen_opt': gen_opt.state_dict(),
                    'disc_A': disc_A.state_dict(),
                    'disc_A_opt': disc_A_opt.state_dict(),
                    'disc_B': disc_B.state_dict(),
                    'disc_B_opt': disc_B_opt.state_dict(),
                }, f'cycleGAN_{cur_step}.pth')
        cur_step += 1

train()
```

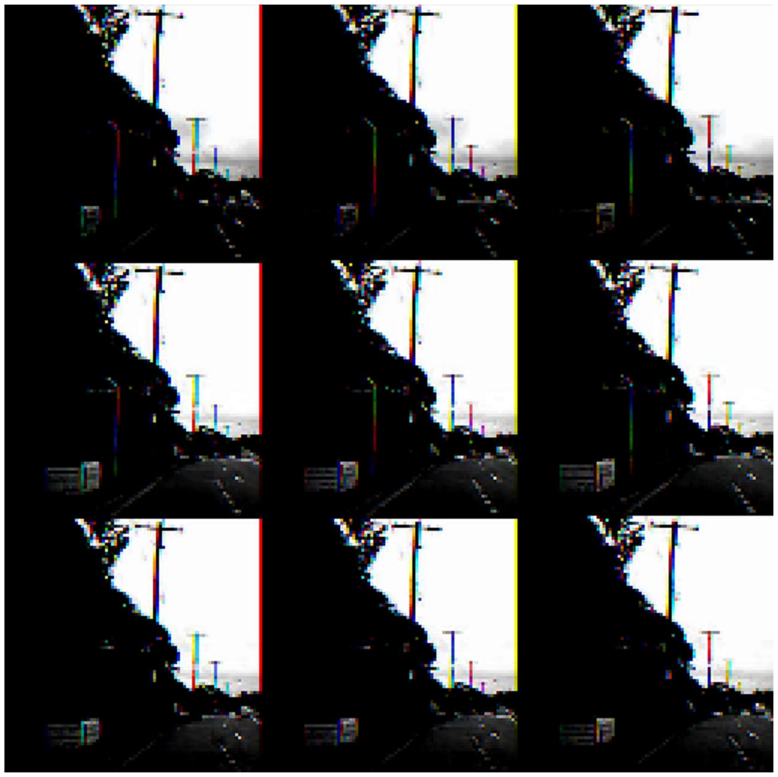
INPUT AND OUTPUT IMAGES:

DAY-NIGHT IMAGE TRANSLATION:

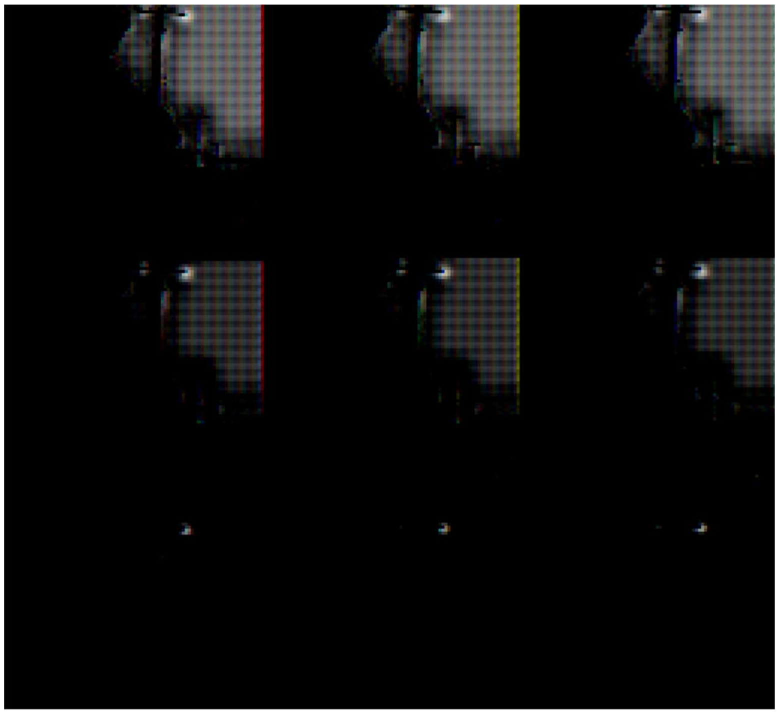
INPUT 1: SAMPLE DAY IMAGE 1



OUTPUT 1: DAY IMAGE 1 TRANSLATED INTO NIGHT IMAGE

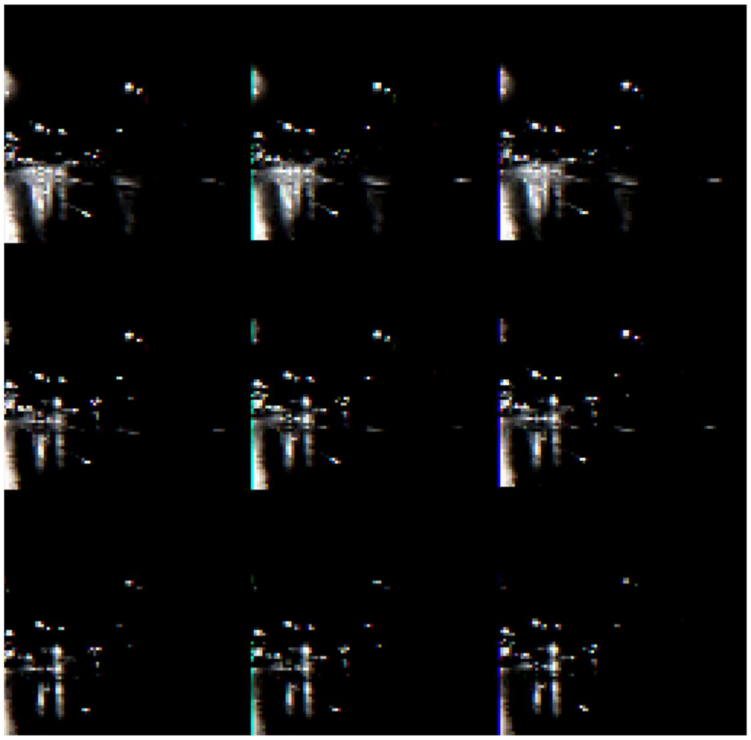


INPUT 2: SAMPLE DAY IMAGE 2

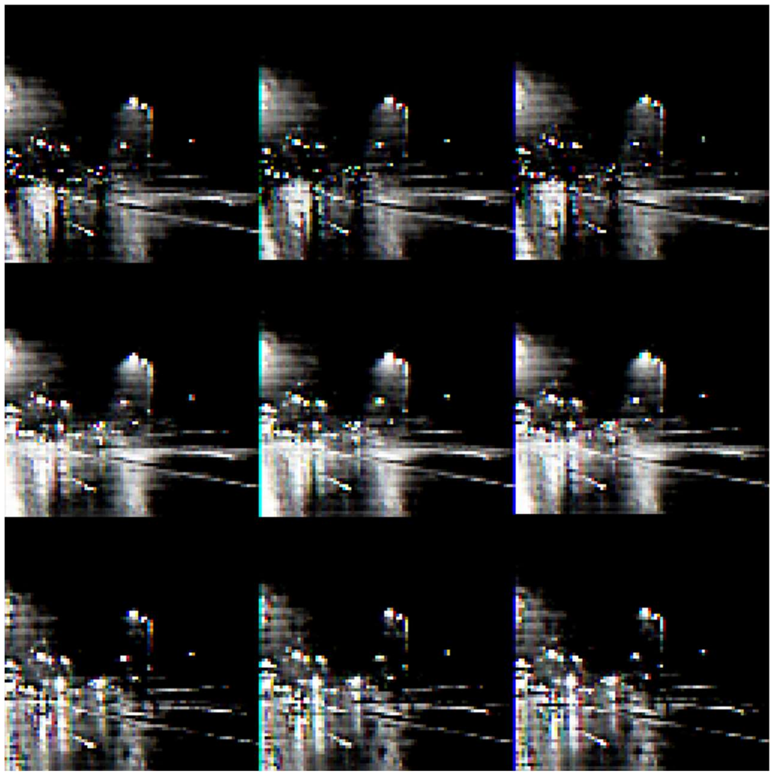


OUTPUT 2: DAY IMAGE 2 TRANSLATED INTO NIGHT IMAGE

NIGHT TO DAY IMAGE TRANSLATION:

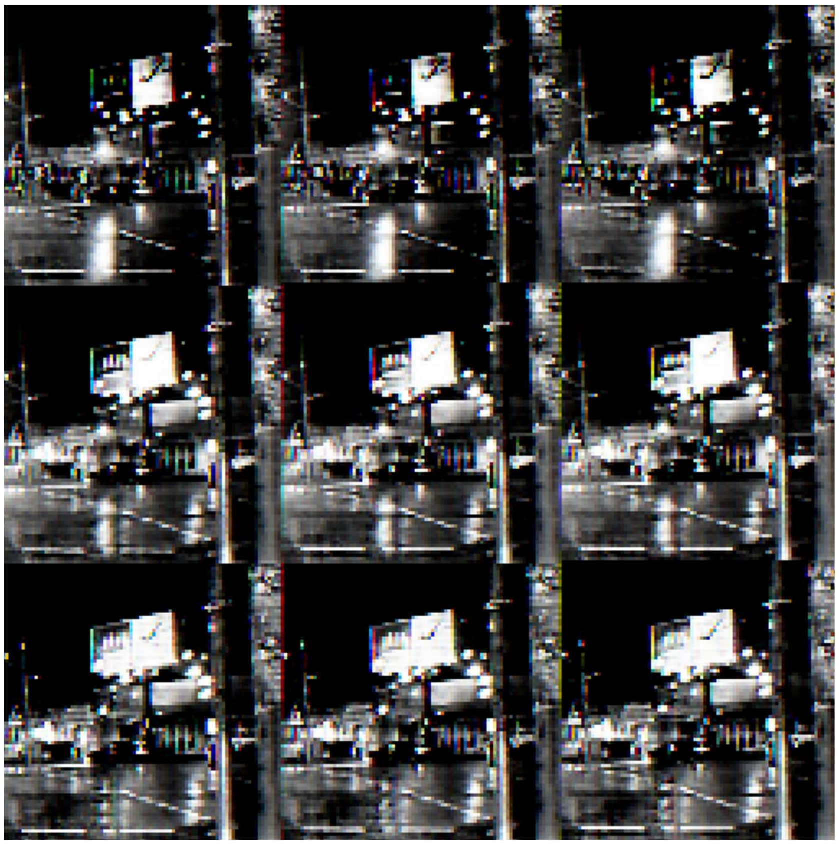


INPUT 3: SAMPLE NIGHT IMAGE 1



OUTPUT 3: NIGHT IMAGE 1 TRANSLATED INTO DAY IMAGE

INPUT 4: SAMPLE NIGHT IMAGE 2



OUTPUT 4: NIGHT IMAGE 2 TRANSLATED INTO DAY IMAGE