MACHINE LEARNING

LAB PROJECT SUBMISSION

SUBMITTED TO : Dr. Simran Setia

SUBMITTED BY :

NAME	ROLL NO
Pranav Sharma	102116090

INDEX

INDEX OF THE PROJECT

INTRODUCTION

Classical Art Converter

OVERVIEW

This project employs CycleGAN, a deep learning model, to transform images into the stylistic representation of a specific artist. The objective is to mimic the artistic style of a chosen painter and apply it to arbitrary input images. By leveraging the power of generative adversarial networks (GANs) and cycle consistency, the model aims to learn the nuances of the artist's unique visual language and reproduce them faithfully in the generated outputs.

OBJECTIVE

The primary goal of this project is to enable users to seamlessly translate any image into the distinctive artistic style of a chosen painter. By harnessing the capabilities of CycleGAN, the model seeks to capture the essence of the artist's brushstrokes, color palette, and overall aesthetic, providing a creative tool for users to explore and experiment with diverse artistic expressions.

APPROACH

The project employs the CycleGAN architecture, consisting of two generators and discriminators, to learn the mapping between input images and the target artist's style. The cycle-consistency loss ensures that the transformation is reversible, maintaining content integrity. Training involves optimizing these networks on a curated dataset, emphasizing the importance of both the source and target artistic styles in achieving accurate and visually appealing results.

KEY COMPONENTS:

Data Collection: The dataset comprises a collection of images representing the artistic style of the chosen painter, curated from sources such as online repositories. This dataset enables the model to learn the intricate details of the artist's unique style, facilitating a more robust and generalized transformation process.

Training and Evaluation: Rigorous model training using labeled datasets, followed by meticulous evaluation to measure accuracy, precision, and recall, ensuring the model's efficacy in correctly identifying various yoga postures.

base_model.py

```python
import os
import torch
from collections import OrderedDict
from abc import ABC, abstractmethod
from . import networks


class BaseModel(ABC):

    def __init__(self, opt):

        self.opt = opt
        self.gpu_ids = opt.gpu_ids
        self.isTrain = opt.isTrain
        self.device = torch.device('cuda:{}'.format(self.gpu_ids[0])) if self.gpu_ids else torch.device('cpu')  # get device name: CPU or
        self.save_dir = os.path.join(opt.checkpoints_dir, opt.name)  # save all the checkpoints to save_dir
        if opt.preprocess != 'scale_width':  # with [scale_width], input images might have different sizes, which hurts the performance o
            torch.backends.cudnn.benchmark = True
        self.loss_names = []
        self.model_names = []
        self.visual_names = []
        self.optimizers = []
        self.image_paths = []
        self.metric = 0  # used for learning rate policy 'plateau'

    @staticmethod
    def modify_commandline_options(parser, is_train):
```

class BaseModel:

```
This class is an abstract base class (ABC) for models.
    To create a subclass, you need to implement the following five functions:
        -- <__init__>:                      initialize the class; first call
BaseModel.__init__(self, opt).
        -- <set_input>:                     unpack data from dataset and apply
preprocessing.
        -- <forward>:                       produce intermediate results.
        -- <optimize_parameters>:           calculate losses, gradients, and
update network weights.
        -- <modify_commandline_options>:    (optionally) add model-specific
options and set default options.
```

__init__ function:

```
Initialize the BaseModel class.

        Parameters:
            opt (Option class)-- stores all the experiment flags; needs to be a
subclass of BaseOptions

        When creating your custom class, you need to implement your own
initialization.
        In this function, you should first call <BaseModel.__init__(self, opt)>
        Then, you need to define four lists:
            -- self.loss_names (str list):          specify the training losses
that you want to plot and save.
```

```
            -- self.model_names (str list):         define networks used in our
training.
            -- self.visual_names (str list):        specify the images that you
want to display and save.
            -- self.optimizers (optimizer list):    define and initialize
optimizers. You can define one optimizer for each network. If two networks are
updated at the same time, you can use itertools.chain to group them. See
cycle_gan_model.py for an example.
```

```python
    @staticmethod
    def modify_commandline_options(parser, is_train):

        return parser

    @abstractmethod
    def set_input(self, input):

        pass

    @abstractmethod
    def forward(self):
        pass

    @abstractmethod
    def optimize_parameters(self):
        pass

    def setup(self, opt):

        if self.isTrain:
            self.schedulers = [networks.get_scheduler(optimizer, opt) for optimizer in self.optimizers]
        if not self.isTrain or opt.continue_train:
            load_suffix = 'iter_%d' % opt.load_iter if opt.load_iter > 0 else opt.epoch
            self.load_networks(load_suffix)
        self.print_networks(opt.verbose)

    def eval(self):
        for name in self.model_names:
            if isinstance(name, str):
                net = getattr(self, 'net' + name)
                net.eval()

    def test(self):

        with torch.no_grad():
            self.forward()
```

Modify_commandline_options function:

```
Add new model-specific options, and rewrite default values for existing options.

        Parameters:
            parser          -- original option parser
            is_train (bool) -- whether training phase or test phase. You can use
this flag to add training-specific or test-specific options.

        Returns:
            the modified parser.
```

Set_input function:

```
Unpack input data from the dataloader and perform necessary pre-processing
steps.

        Parameters:
            input (dict): includes the data itself and its metadata information.
```

forward function:

```
Run forward pass; called by both functions <optimize_parameters> and <test>.
```

setup function:

```
Load and print networks; create schedulers

        Parameters:
            opt (Option class) -- stores all the experiment flags; needs to be a
subclass of BaseOptions
```

eval function:

```
Make models eval mode during test time
```

test function:

```
Forward function used in test time.

        This function wraps <forward> function in no_grad() so we don't save
intermediate steps for backprop
        It also calls <compute_visuals> to produce additional visualization
results
```

```python
    def test(self):

        with torch.no_grad():
            self.forward()
            self.compute_visuals()

    def compute_visuals(self):
        pass

    def get_image_paths(self):
        return self.image_paths

    def update_learning_rate(self):
        old_lr = self.optimizers[0].param_groups[0]['lr']
        for scheduler in self.schedulers:
            if self.opt.lr_policy == 'plateau':
                scheduler.step(self.metric)
            else:
                scheduler.step()

        lr = self.optimizers[0].param_groups[0]['lr']
        print('learning rate %.7f -> %.7f' % (old_lr, lr))

    def get_current_visuals(self):
        visual_ret = OrderedDict()
        for name in self.visual_names:
            if isinstance(name, str):
                visual_ret[name] = getattr(self, name)
        return visual_ret

    def get_current_losses(self):
        errors_ret = OrderedDict()
        for name in self.loss_names:
            if isinstance(name, str):
                errors_ret[name] = float(getattr(self, 'loss_' + name))  # float(...) works for both scalar tensor and float number
        return errors_ret
```

compute_visuals function:

```
Calculate additional output images for visdom and HTML visualization
```

get_image_paths function:

```
Return image paths that are used to load current data
```

update_learning_rate function:

```
Update learning rates for all the networks; called at the end of every epoch
```

get_current_visuals function:

```
Return visualization images. train.py will display these images with visdom, and
save the images to a HTML
```

get_current_losses function:

```
Return traning losses / errors. train.py will print out these errors on console,
and save them to a file
```

```python
        errors_ret = OrderedDict()
        for name in self.loss_names:
            if isinstance(name, str):
                errors_ret[name] = float(getattr(self, 'loss_' + name))  # float(...) works for b
        return errors_ret

    def save_networks(self, epoch):

        for name in self.model_names:
            if isinstance(name, str):
                save_filename = '%s_net_%s.pth' % (epoch, name)
                save_path = os.path.join(self.save_dir, save_filename)
                net = getattr(self, 'net' + name)

                if len(self.gpu_ids) > 0 and torch.cuda.is_available():
                    torch.save(net.module.cpu().state_dict(), save_path)
                    net.cuda(self.gpu_ids[0])
                else:
                    torch.save(net.cpu().state_dict(), save_path)

    def __patch_instance_norm_state_dict(self, state_dict, module, keys, i=0):
        key = keys[i]
        if i + 1 == len(keys):  # at the end, pointing to a parameter/buffer
            if module.__class__.__name__.startswith('InstanceNorm') and \
                    (key == 'running_mean' or key == 'running_var'):
                if getattr(module, key) is None:
                    state_dict.pop('.'.join(keys))
            if module.__class__.__name__.startswith('InstanceNorm') and \
               (key == 'num_batches_tracked'):
                state_dict.pop('.'.join(keys))
        else:
            self.__patch_instance_norm_state_dict(state_dict, getattr(module, key), keys, i + 1)

    def load_networks(self, epoch):
```

save_networks function:

```
Save all the networks to the disk.

        Parameters:
            epoch (int) -- current epoch; used in the file name '%s_net_%s.pth'
% (epoch, name)
```

__patch_instance_norm_state_dict function:

```
Fix InstanceNorm checkpoints incompatibility (prior to 0.4)
```

```python
            else:
                self.__patch_instance_norm_state_dict(state_dict, getattr(module, key), keys, i + 1)

    def load_networks(self, epoch):

        for name in self.model_names:
            if isinstance(name, str):
                load_filename = '%s_net_%s.pth' % (epoch, name)
                load_path = os.path.join(self.save_dir, load_filename)
                net = getattr(self, 'net' + name)
                if isinstance(net, torch.nn.DataParallel):
                    net = net.module
                print('loading the model from %s' % load_path)
                # if you are using PyTorch newer than 0.4 (e.g., built from
                # GitHub source), you can remove str() on self.device
                state_dict = torch.load(load_path, map_location=str(self.device))
                if hasattr(state_dict, '_metadata'):
                    del state_dict._metadata

                # patch InstanceNorm checkpoints prior to 0.4
                for key in list(state_dict.keys()):  # need to copy keys here because we mutate in loop
                    self.__patch_instance_norm_state_dict(state_dict, net, key.split('.'))
                net.load_state_dict(state_dict)

    def print_networks(self, verbose):

        print('---------- Networks initialized -------------')
        for name in self.model_names:
            if isinstance(name, str):
                net = getattr(self, 'net' + name)
                num_params = 0
                for param in net.parameters():
                    num_params += param.numel()
                if verbose:
                    print(net)
                print('[Network %s] Total number of parameters : %.3f M' % (name, num_params / 1e6))
        print('-----------------------------------------------')
```

load_network function:

```
Load all the networks from the disk.

        Parameters:
            epoch (int) -- current epoch; used in the file name '%s_net_%s.pth' % (epoch, name)
```

print_network function:

```
Print the total number of parameters in the network and (if verbose) network architecture

        Parameters:
            verbose (bool) -- if verbose: print the network architecture
```

```
                print('[Network %s] Total number of parameters : %.3f M' % (name, num_params / 1e6))
        print('-----------------------------------------------')

    def set_requires_grad(self, nets, requires_grad=False):

        if not isinstance(nets, list):
            nets = [nets]
        for net in nets:
            if net is not None:
                for param in net.parameters():
                    param.requires_grad = requires_grad
```

set_requires_grad function:

```
Set requies_grad=Fasle for all the networks to avoid unnecessary computations
        Parameters:
            nets (network list)    -- a list of networks
            requires_grad (bool)   -- whether the networks require gradients or
not
```

colorization_model.py

```python
from .pix2pix_model import Pix2PixModel
import torch
from skimage import color  # used for lab2rgb
import numpy as np


class ColorizationModel(Pix2PixModel):

    @staticmethod
    def modify_commandline_options(parser, is_train=True):

        Pix2PixModel.modify_commandline_options(parser, is_train)
        parser.set_defaults(dataset_mode='colorization')
        return parser

    def __init__(self, opt):

        # reuse the pix2pix model
        Pix2PixModel.__init__(self, opt)
        # specify the images to be visualized.
        self.visual_names = ['real_A', 'real_B_rgb', 'fake_B_rgb']

    def lab2rgb(self, L, AB):

        AB2 = AB * 110.0
        L2 = (L + 1.0) * 50.0
        Lab = torch.cat([L2, AB2], dim=1)
        Lab = Lab[0].data.cpu().float().numpy()
        Lab = np.transpose(Lab.astype(np.float64), (1, 2, 0))
        rgb = color.lab2rgb(Lab) * 255
        return rgb

    def compute_visuals(self):
        self.real_B_rgb = self.lab2rgb(self.real_A, self.real_B)
        self.fake_B_rgb = self.lab2rgb(self.real_A, self.fake_B)

```

class ColorizationModel:

```
This is a subclass of Pix2PixModel for image colorization (black & white image
-> colorful images).


    The model training requires '-dataset_model colorization' dataset.
    It trains a pix2pix model, mapping from L channel to ab channels in Lab
color space.
    By default, the colorization dataset will automatically set '--input_nc 1'
and '--output_nc 2'.
```

modify_commandline_options function:

```
Add new dataset-specific options, and rewrite default values for existing
options.

        Parameters:
            parser          -- original option parser
            is_train (bool) -- whether training phase or test phase. You can use
this flag to add training-specific or test-specific options.

        Returns:
            the modified parser.

        By default, we use 'colorization' dataset for this model.
        See the original pix2pix paper (https://arxiv.org/pdf/1611.07004.pdf)
and colorization results (Figure 9 in the paper)
```

__init__ function:

```
Initialize the class.

        Parameters:
            opt (Option class)-- stores all the experiment flags; needs to be a
subclass of BaseOptions

        For visualization, we set 'visual_names' as 'real_A' (input real image),
        'real_B_rgb' (ground truth RGB image), and 'fake_B_rgb' (predicted RGB
image)
        We convert the Lab image 'real_B' (inherited from Pix2pixModel) to a RGB
image 'real_B_rgb'.
        we convert the Lab image 'fake_B' (inherited from Pix2pixModel) to a RGB
image 'fake_B_rgb'.
```

lab2rgb function:

```
Convert an Lab tensor image to a RGB numpy output
        Parameters:
            L  (1-channel tensor array): L channel images (range: [-1, 1], torch
tensor array)
            AB (2-channel tensor array):  ab channel images (range: [-1, 1],
torch tensor array)

        Returns:
            rgb (RGB numpy image): rgb output images  (range: [0, 255], numpy
array)
```

compute_visuals function:

```
Calculate additional output images for visdom and HTML visualization
```

cycle_gan_model.py

```python
import torch
import itertools
from util.image_pool import ImagePool
from .base_model import BaseModel
from . import networks


class CycleGANModel(BaseModel):

    @staticmethod
    def modify_commandline_options(parser, is_train=True):

        parser.set_defaults(no_dropout=True)  # default CycleGAN did not use dropout
        if is_train:
            parser.add_argument('--lambda_A', type=float, default=10.0, help='weight for cycle loss (A -> B -> A)')
            parser.add_argument('--lambda_B', type=float, default=10.0, help='weight for cycle loss (B -> A -> B)')
            parser.add_argument('--lambda_identity', type=float, default=0.5, help='use identity mapping. Setting lambda_identity other

        return parser

    def __init__(self, opt):

        BaseModel.__init__(self, opt)
        # specify the training losses you want to print out. The training/test scripts will call <BaseModel.get_current_losses>
        self.loss_names = ['D_A', 'G_A', 'cycle_A', 'idt_A', 'D_B', 'G_B', 'cycle_B', 'idt_B']
        # specify the images you want to save/display. The training/test scripts will call <BaseModel.get_current_visuals>
        visual_names_A = ['real_A', 'fake_B', 'rec_A']
        visual_names_B = ['real_B', 'fake_A', 'rec_B']
        if self.isTrain and self.opt.lambda_identity > 0.0:  # if identity loss is used, we also visualize idt_B=G_A(B) ad idt_A=G_A(B)
            visual_names_A.append('idt_B')
            visual_names_B.append('idt_A')

        self.visual_names = visual_names_A + visual_names_B  # combine visualizations for A and B
        # specify the models you want to save to the disk. The training/test scripts will call <BaseModel.save_networks> and <BaseModel.
        if self.isTrain:
            self.model_names = ['G_A', 'G_B', 'D_A', 'D_B']
```

class CycleGANModel:

```
This class implements the CycleGAN model, for learning image-to-image
translation without paired data.

    The model training requires '--dataset_mode unaligned' dataset.
    By default, it uses a '--netG resnet_9blocks' ResNet generator,
    a '--netD basic' discriminator (PatchGAN introduced by pix2pix),
    and a least-square GANs objective ('--gan_mode lsgan').

    CycleGAN paper: https://arxiv.org/pdf/1703.10593.pdf
```

modify_commandline_options function:

```
Add new dataset-specific options, and rewrite default values for existing
options.

        Parameters:
            parser          -- original option parser
            is_train (bool) -- whether training phase or test phase. You can use
this flag to add training-specific or test-specific options.

        Returns:
```

```
            the modified parser.

        For CycleGAN, in addition to GAN losses, we introduce lambda_A,
lambda_B, and lambda_identity for the following losses.
        A (source domain), B (target domain).
        Generators: G_A: A -> B; G_B: B -> A.
        Discriminators: D_A: G_A(A) vs. B; D_B: G_B(B) vs. A.
        Forward cycle loss:  lambda_A * ||G_B(G_A(A)) - A|| (Eqn. (2) in the
paper)
        Backward cycle loss: lambda_B * ||G_A(G_B(B)) - B|| (Eqn. (2) in the
paper)
        Identity loss (optional): lambda_identity * (||G_A(B) - B|| * lambda_B +
||G_B(A) - A|| * lambda_A) (Sec 5.2 "Photo generation from paintings" in the
paper)
        Dropout is not used in the original CycleGAN paper.
```

__init__ function:

```
Initialize the CycleGAN class.

        Parameters:
            opt (Option class)-- stores all the experiment flags; needs to be a
subclass of BaseOptions
```

```
        if self.isTrain:
            self.model_names = ['G_A', 'G_B', 'D_A', 'D_B']
        else:  # during test time, only load Gs
            self.model_names = ['G_A', 'G_B']

        # define networks (both Generators and discriminators)
        # The naming is different from those used in the paper.
        # Code (vs. paper): G_A (G), G_B (F), D_A (D_Y), D_B (D_X)
        self.netG_A = networks.define_G(opt.input_nc, opt.output_nc, opt.ngf, opt.netG, opt.norm,
                                        not opt.no_dropout, opt.init_type, opt.init_gain, self.gpu_ids)
        self.netG_B = networks.define_G(opt.output_nc, opt.input_nc, opt.ngf, opt.netG, opt.norm,
                                        not opt.no_dropout, opt.init_type, opt.init_gain, self.gpu_ids)

        if self.isTrain:  # define discriminators
            self.netD_A = networks.define_D(opt.output_nc, opt.ndf, opt.netD,
                                            opt.n_layers_D, opt.norm, opt.init_type, opt.init_gain, self.gpu_ids)
            self.netD_B = networks.define_D(opt.input_nc, opt.ndf, opt.netD,
                                            opt.n_layers_D, opt.norm, opt.init_type, opt.init_gain, self.gpu_ids)

        if self.isTrain:
            if opt.lambda_identity > 0.0:  # only works when input and output images have the same number of channels
                assert(opt.input_nc == opt.output_nc)
            self.fake_A_pool = ImagePool(opt.pool_size)  # create image buffer to store previously generated images
            self.fake_B_pool = ImagePool(opt.pool_size)  # create image buffer to store previously generated images
            # define loss functions              See Real World Examples From GitHub
            self.criterionGAN = networks.                                      )  # define GAN loss.
            self.criterionCycle = torch.n (function) L1Loss: Any
            self.criterionIdt = torch.nn.L1Loss()
            # initialize optimizers; schedulers will be automatically created by function <BaseModel.setup>.
            self.optimizer_G = torch.optim.Adam(itertools.chain(self.netG_A.parameters(), self.netG_B.parameters()), lr=opt.lr, betas=(op
            self.optimizer_D = torch.optim.Adam(itertools.chain(self.netD_A.parameters(), self.netD_B.parameters()), lr=opt.lr, betas=(op
            self.optimizers.append(self.optimizer_G)
            self.optimizers.append(self.optimizer_D)

    def set_input(self, input):
```

```python
    def set_input(self, input):

        AtoB = self.opt.direction == 'AtoB'
        self.real_A = input['A' if AtoB else 'B'].to(self.device)
        self.real_B = input['B' if AtoB else 'A'].to(self.device)
        self.image_paths = input['A_paths' if AtoB else 'B_paths']

    def forward(self):
        self.fake_B = self.netG_A(self.real_A)  # G_A(A)
        self.rec_A = self.netG_B(self.fake_B)   # G_B(G_A(A))
        self.fake_A = self.netG_B(self.real_B)  # G_B(B)
        self.rec_B = self.netG_A(self.fake_A)   # G_A(G_B(B))

    def backward_D_basic(self, netD, real, fake):

        # Real
        pred_real = netD(real)
        loss_D_real = self.criterionGAN(pred_real, True)
        # Fake
        pred_fake = netD(fake.detach())
        loss_D_fake = self.criterionGAN(pred_fake, False)
        # Combined loss and calculate gradients
        loss_D = (loss_D_real + loss_D_fake) * 0.5
        loss_D.backward()
        return loss_D

    def backward_D_A(self):
        fake_B = self.fake_B_pool.query(self.fake_B)
        self.loss_D_A = self.backward_D_basic(self.netD_A, self.real_B, fake_B)

    def backward_D_B(self):
        fake_A = self.fake_A_pool.query(self.fake_A)
        self.loss_D_B = self.backward_D_basic(self.netD_B, self.real_A, fake_A)

    def backward_G(self):
        lambda_idt = self.opt.lambda_identity
        lambda_A = self.opt.lambda_A
```

set_input function:

```
Unpack input data from the dataloader and perform necessary pre-processing
steps.

        Parameters:
            input (dict): include the data itself and its metadata information.

        The option 'direction' can be used to swap domain A and domain B.
```

forward function:

```
Run forward pass; called by both functions <optimize_parameters> and <test>.
```

backward_D_basic function:

```
Calculate GAN loss for the discriminator

        Parameters:
            netD (network)      -- the discriminator D
            real (tensor array) -- real images
            fake (tensor array) -- images generated by a generator

        Return the discriminator loss.
        We also call loss_D.backward() to calculate the gradients.
```

backward_D_A function:

```
Calculate GAN loss for discriminator D_A
```

backward_D_B function:

```
Calculate GAN loss for discriminator D_B
```

backward_G function:

```
Calculate the loss for generators G_A and G_B
```

```python
    def backward_G(self):
        lambda_idt = self.opt.lambda_identity
        lambda_A = self.opt.lambda_A
        lambda_B = self.opt.lambda_B
        # Identity loss
        if lambda_idt > 0:
            # G_A should be identity if real_B is fed: ||G_A(B) - B||
            self.idt_A = self.netG_A(self.real_B)
            self.loss_idt_A = self.criterionIdt(self.idt_A, self.real_B) * lambda_B * lambda_idt
            # G_B should be identity if real_A is fed: ||G_B(A) - A||
            self.idt_B = self.netG_B(self.real_A)
            self.loss_idt_B = self.criterionIdt(self.idt_B, self.real_A) * lambda_A * lambda_idt
        else:
            self.loss_idt_A = 0
            self.loss_idt_B = 0

        # GAN loss D_A(G_A(A))
        self.loss_G_A = self.criterionGAN(self.netD_A(self.fake_B), True)
        # GAN loss D_B(G_B(B))
        self.loss_G_B = self.criterionGAN(self.netD_B(self.fake_A), True)
        # Forward cycle loss || G_B(G_A(A)) - A||
        self.loss_cycle_A = self.criterionCycle(self.rec_A, self.real_A) * lambda_A
        # Backward cycle loss || G_A(G_B(B)) - B||
        self.loss_cycle_B = self.criterionCycle(self.rec_B, self.real_B) * lambda_B
        # combined loss and calculate gradients
        self.loss_G = self.loss_G_A + self.loss_G_B + self.loss_cycle_A + self.loss_cycle_B + self.loss_idt_A + self.loss_idt_B
        self.loss_G.backward()
```

```python
    def optimize_parameters(self):
        # forward
        self.forward()      # compute fake images and reconstruction images.
        # G_A and G_B
        self.set_requires_grad([self.netD_A, self.netD_B], False)  # Ds require no gradients when optimizing Gs
        self.optimizer_G.zero_grad()  # set G_A and G_B's gradients to zero
        self.backward_G()             # calculate gradients for G_A and G_B
        self.optimizer_G.step()       # update G_A and G_B's weights
        # D_A and D_B
        self.set_requires_grad([self.netD_A, self.netD_B], True)
        self.optimizer_D.zero_grad()   # set D_A and D_B's gradients to zero
        self.backward_D_A()      # calculate gradients for D_A
        self.backward_D_B()      # calculate graidents for D_B
        self.optimizer_D.step()  # update D_A and D_B's weights
```

optimize_parameters function:

```
Calculate losses, gradients, and update network weights; called in every
training iteration
```

RESULTS

RESULTS OF THE PROJECT

MODEL PERFORMANCE AND TRAINING

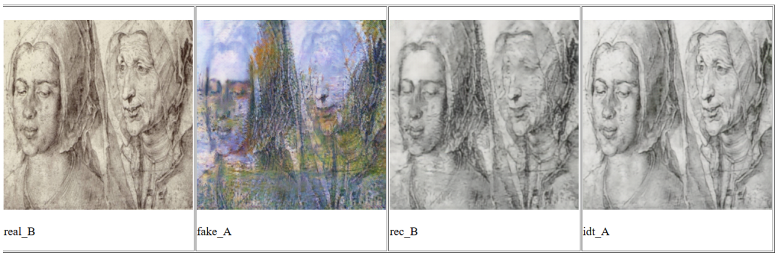
Visual Comparison:

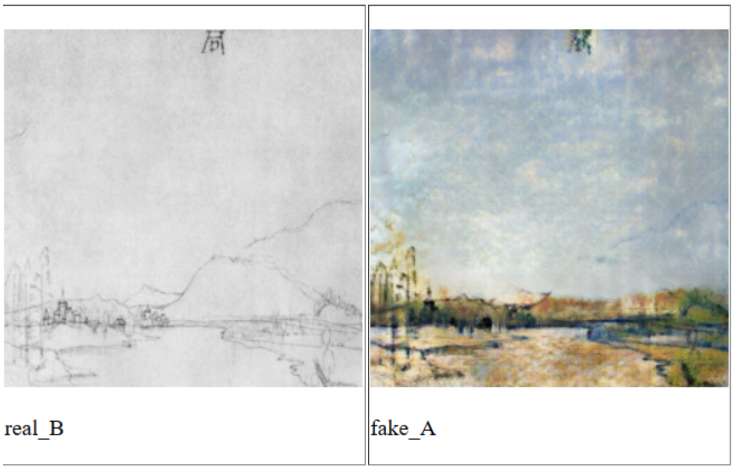
Generated vs. Ground Truth:

epoch [174]

Domain-Specific Evaluation:

Edges and Color Correction:





Highlights the effectiveness of discriminators specialized in evaluating edges and color correctness.

Provide examples showcasing enhanced details and color accuracy.

Style Limitations: Classical art generators using CycleGANs may struggle to handle diverse or abstract artistic styles, leading to challenges in faithfully reproducing certain genres.

Detail Loss: Fine details in high-resolution artworks may be lost during the image translation process, affecting the overall fidelity of the generated images.

CONCLUSION

In conclusion, this project successfully leverages CycleGAN for artistic style transfer, providing a valuable tool for users to explore and apply the distinctive styles of renowned painters to their own images. The project contributes to the intersection of deep learning and creative expression, opening avenues for further research and applications in the domain of computer-assisted art creation.