STYLE TRANSFER USING CYCLE GAN

A major project report submitted to CUSAT in partial fulfilment for the award of degree of

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE AND ENGINEERING

Submitted by

Adithya S Kumar (12170200)
Anand Pavithran (12170207)
Jomon Joseph(12170222)
Muhammed Azhar Juman PV(12170228)
Shaun Saju (12170237)

Under the Guidance of

Mrs. Sreeja Nair M P & Dr. Preetha Mathew

Department of Computer Science &Engineering

COCHIN UNIVERSITY COLLEGE OF ENGINEERING KUTTANADU

June 2020

COCHIN UNIVERSITY COLLEGE OF ENGINEERING KUTTANADU ALAPPUZHA

BONAFIDE CERTIFICATE

This is to certify that the project report entitled "**STYLE TRANSFER USING CYCLE GAN"** has been submitted by **Adithya S Kumar, Anand Pavithran, Jomon Joseph, Muhammed Azhar Juman PV, Shaun Saju** in partial fulfilment of the requirements for the award of degree of Bachelor of Technology in Computer Science & Engineering is a bonafide record of the work carried out by them under my guidance and supervision at Cochin University college of Engineering Kuttanadu, Alappuzha, during the academic 2019-2020.

Mrs. Sreeja Nair M P	**Dr. Preetha Mathew**	**Mrs. Bindu PK**
Assistant Professor, CSE	**Associate Professor, CSE**	**HOD, CSE**

Place: Pulincunnoo

Date: 17-06-2020

ACKNOWLEDGEMENT

Many people have contributed to the success of this Project. Although a single sentence hardly suffices, we would like to thank Almighty God for blessing us with His grace and taking our endeavor to a successful culmination. We are extremely grateful to our principal **Dr. SUNIL KUMAR N**, for his whole hearted cooperation for the success complication of project. We extend our sincere and heart-full thanks to **Mrs. Bindu PK** Head of Department (COMPUTER SCIENCE AND ENGINEERING) for providing us the right ambience for carrying out the work on this project and for the facilities provided to us. We are profoundly indebted to our project guide**, Mrs. Sreeja Nair M P**, Assistant Professor (CSE) and **Dr. Preetha Mathew,** Associate Professor (CSE) for their innumerable acts of timely advice, encouragement and we sincerely express our gratitude to them.

We would like to extend our gratitude to all the staff of the Department of Computer Science And Engineering for the help and support rendered to us. We have benefitted a lot from the feedback and suggestions given to us by them.

ADITHYA S KUMAR, S8 CSE

ANAND PAVITHRAN, S8 CSE

JOMON JOSEPH, S8 CSE

MUHAMMED AZHAR JUMAN PV, S8 CSE

SHAUN SAJU, S8 CSE

ABSTRACT

Computer Vision is an endeavor that attempts to give computers the ability to analyze a scene or an image, much like a human visual system. Neural Networks have been proven to shine in this area, giving much better results and a faster performance than traditional algorithms. Style Transfer is a relatively new technique used to translate the properties of a class of images to another class of images. The goal is to automatically learn the distinct features of an image set and map them to the features of a second set of images, and how to transfer those features. However, for many tasks, paired training data will not be available. Therefore we use a novel architecture called CycleGAN (Cycle-Consistent Generative Adversarial Network) to perform style transfer. CycleGAN's design allows us to just train the network with two sets of images rather than input-output pairs, which makes obtaining of training data that much easier. We train CycleGANs with a variety of training data, analyzing the neural models that give best results under various constraints. Qualitative results are presented on several tasks where paired training data does not exist, including collection style transfer, object transfiguration, season transfer, photo enhancement, etc. Image-to-image translation is also important in the task of domain adaptation. For safety reasons, robots are often trained in simulated environment and using synthesized data. In order for such trained robots to behave well in real life scenarios, one possible approach is to translate real life data, e.g., images, into data similar to what they aretrained with using image-to-image translation techniques.

INDEX

LIST OF SCREENSHOTS

LIST OF DIAGRAMS

1. INTRODUCTION

The way we communicate, pay and travel has evolved over the last several years partly due to advancements in machine learning. In fact, credit card companies use fraud detection algorithms to label transactions as fraudulent or not, Facebook uses facial recognition to detect faces in photos and Tesla uses computer vision to recognize traffic signs. These tasks can be solved using by discriminative models that can separate different samples into different classes. However, this approach requires datasets to be annotated which in some cases can be expensive. Another approach which does not necessarily require annotations uses a generator in combination with a discriminator. The generator is trained to generate data according to a data distribution. In the field of artificial neural networks (ANN), there are several different models which are capable of such tasks, e.g. restricted Boltzmann machines, variational auto encoders or generative adversarial networks (GAN). This project will focus on the cycle-consistent generative adversarial network (CycleGAN) which is a type of GAN. CycleGAN is designed to perform unpaired image-to-image translations. This means that the input and the output of the artificial neural network is an image. In the world of image-to-image translation, there are two variants of training data: the data can be paired or unpaired. In the former case, every input sample is paired with exactly one output sample. In the latter case, there is no such mapping. Colorizing black and white images is an example of a paired image-to-image translation. However, paired data is not always available. Style transfer or image stylification is a problem for which there are, usually, no paired data available. For example, training a model to translate pictures to look like Piccasso paintings requires paired data which are typically not available. An important constraint to such translations is that the semantics of the input must be preserved. For example, a photo of a bridge, when stylized in the style of Van Gogh, should look like a painting of the same bridge. CycleGAN is capable of preserving semantics on unpaired training dat

2. FEASIBILTY STUDY

A feasibility study is a high-level capsule version of the entire system analysis and Design Process. The Study begins by classifying the problem definition. Feasibility is to determine if it's worth doing. Once an acceptance problem definition has been generated, the analyst develops a logical model of the system. A search for alternative is analyzed carefully.

Technical feasibility

Technical feasibility is one of the first studies that must be conducted after the project has been identified. Technical feasibility study includes the hardware and software devices. The required technologies specifications like Tensorflow, Python, OpenCV exists hence this is technically feasible. The hardware required is also available, whether it be machines at our university, cloud computing services.

Operational feasibility

Operational Feasibility is a measure of how well a proposed system work out for the goal and takes advantage of the opportunities identified during scope definition. Our system can be tweaked and modified as the situation calls as the results are instantly visible.

Economic feasibility

The purpose of economic feasibility is to determine the positive economic benefits that include quantification and identification. The system is economically feasible due to the availability of all requirements such as collection of data from open sources online such as Kaggle (which is a website focused on machine learning datasets).

3. REQUIREMENT ANALYSIS

3.1 Software Requirement Specifications

The software requirement specification is the process of analyzing the requirement with the potential goal of improving or modifying it.

Requirement analysis is an important phase during application development. Mainly it contains the analyzing phase of the existing system and its features and also the proposed system and also the proposed system.

Analyzing phase gives advantages and disadvantages of the proposed system, which can avoid all the complexities, inabilities, and the disadvantage of the existing system. The new system requirement is defining during this phase. The requirement of the desired software product is extracted. To design a new system, we need the requirement of the system and the description of the system. Based On the business scenario the software requirement specification document is prepared in this phase. The purpose of this document is to specify the functional requirement of the software that is to be built.

3.2 Purpose

The purpose of this project is to transform the characteristics from one image to another image using Generative adversarial network.

3.3 Scope

CAMERA FILTERS

Several image filters can be developed by training the CycleGAN with various data sets. Such filters are very popular in the consumer market, especially for mobile phone applications.

ROBOTIC VISION

The technology could be used as an interface between a robot’s visual processing system and the real world. A robot can be trained on a simplified simulation of the world, and the CycleGAN can be trained to translate the real world into the simplified style that can be parsed easily.

NOVEL CGI EFFECTS

Style transfer can be used for CGI effects in movies. CGI in movies is a time consuming and expensive process, and style transfer can make the process much less cumbersome, either used directly or as a starting point for a CGI artist.

3.4 Hardware requirements:

- System with a good multi-core CPU(Intel preferred) and GPU (NVIDIA with CUDA support).
- RAM – minimum 4 Gb
- Internal memory: 10 Gb

3.5 Software requirements:

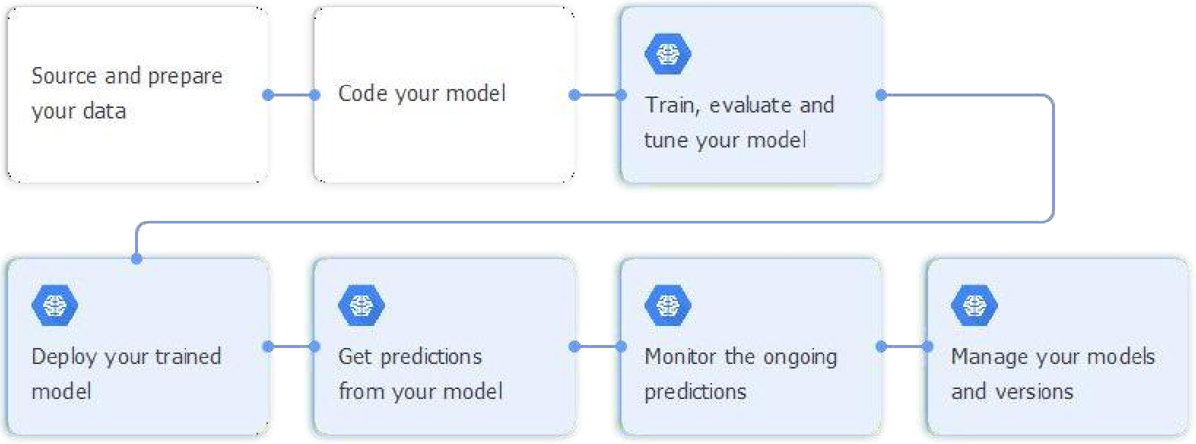
- Tensorflow
- Pytorch
- Python as primary programming language.
- Image datasets.

3.6 Functional Requirements:

- Satisfactory quality in translation of images.
- Feasible performance time.
- Cross-Platform implementation.

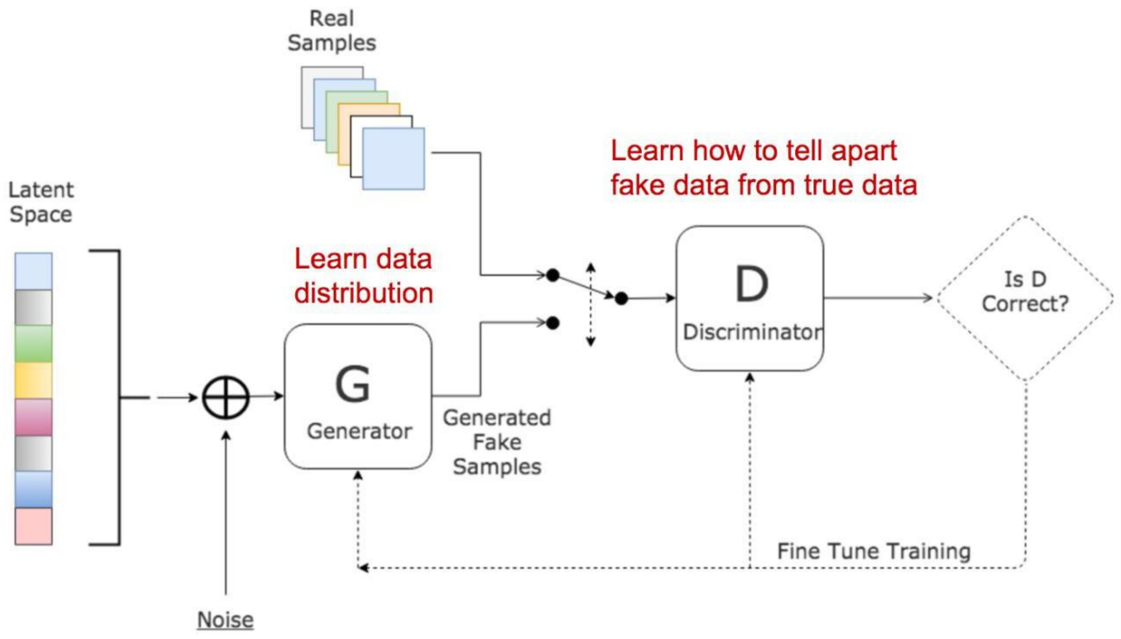
4. DESIGN

4.1 Workflow of a general Machine Learning Project:


Source and prepare your data
Code your model
Train, evaluate and tune your model
Deploy your trained model
Get predictions from your model
Monitor the ongoing predictions
Manage your models and versions

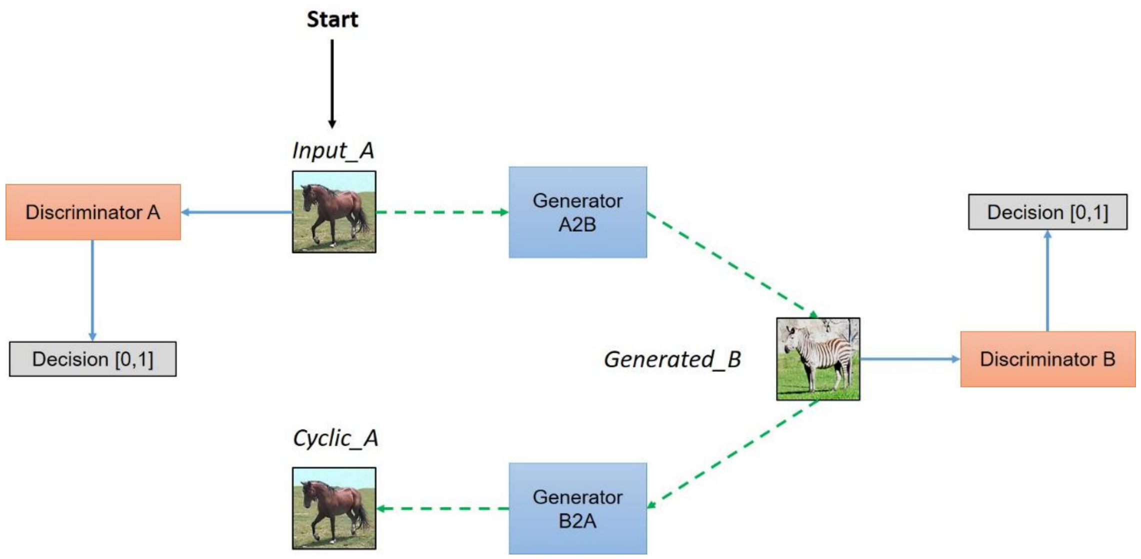
Since this is primarily a machine learning project, it will consist of two phases: Training, and Deployment. The training phase is where the neural networks learn from the give datasets. The deployment phase is when the neural networks are deployed as an application to produce synthesized images.

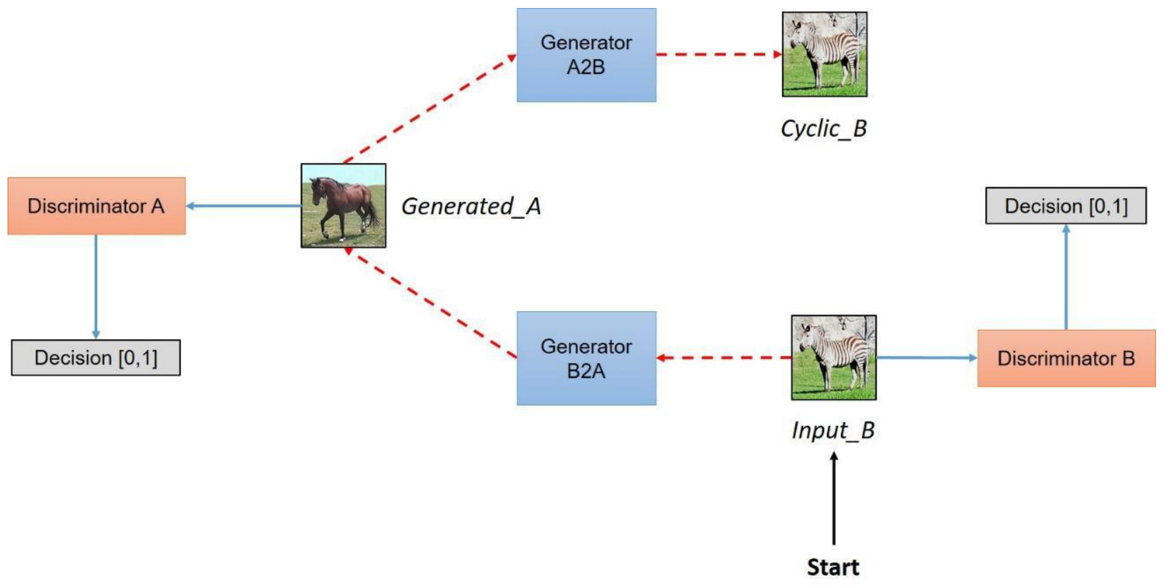
4.2 Activity Diagram for GAN:



In Generative Adversarial Networks (GAN), two neural networks contest with each other in a game (in the sense of game theory, often but not always in the form of a zero-sum game). Given a training set, this technique learns to generate new data with the same statistics as the training set. For example, a GAN trained on photographs can generate new photographs that look at least superficially authentic to human observers, having many realistic characteristics. The *generative* network generates candidates while the discriminative network evaluates them. The contest operates in terms of data distributions. Typically, the generative network learns to map from a latent space to a data distribution of interest, while the discriminative network distinguishes candidates produced by the generator from the true data distribution. The generative network's training objective is to increase the error rate of the discriminative network (i.e., "fool" the discriminator network by producing novel candidates that the discriminator thinks are not synthesized (are part of the true data distribution)).

4.3 CycleGAN architecture:

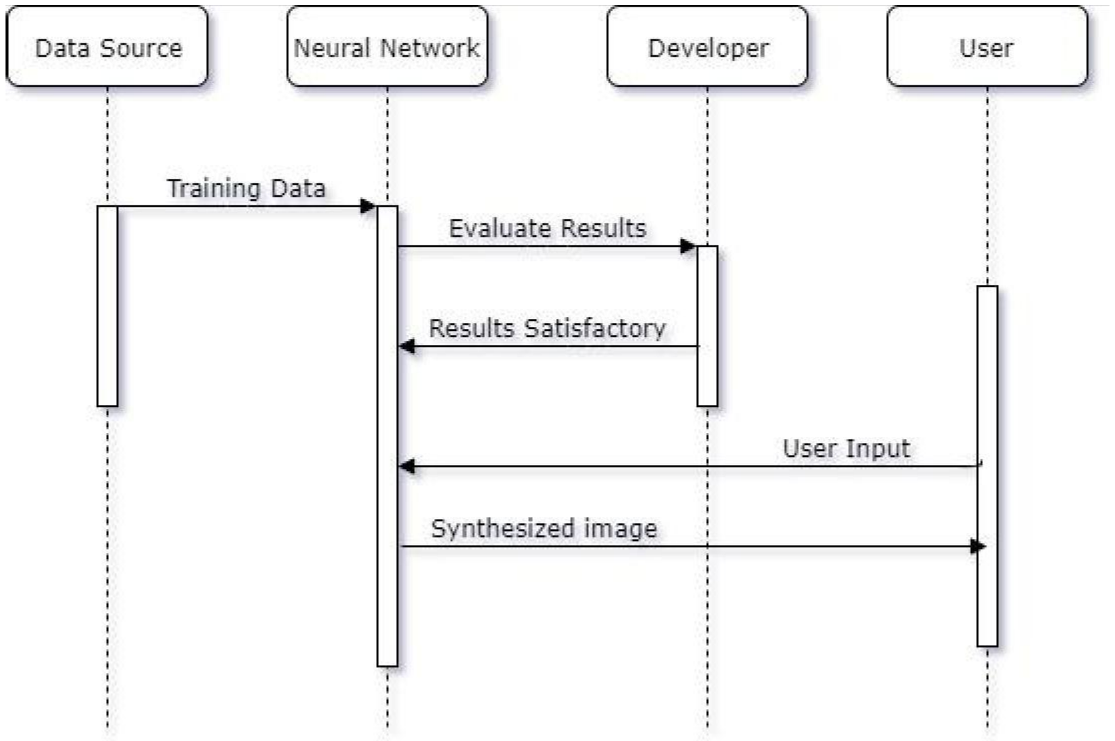




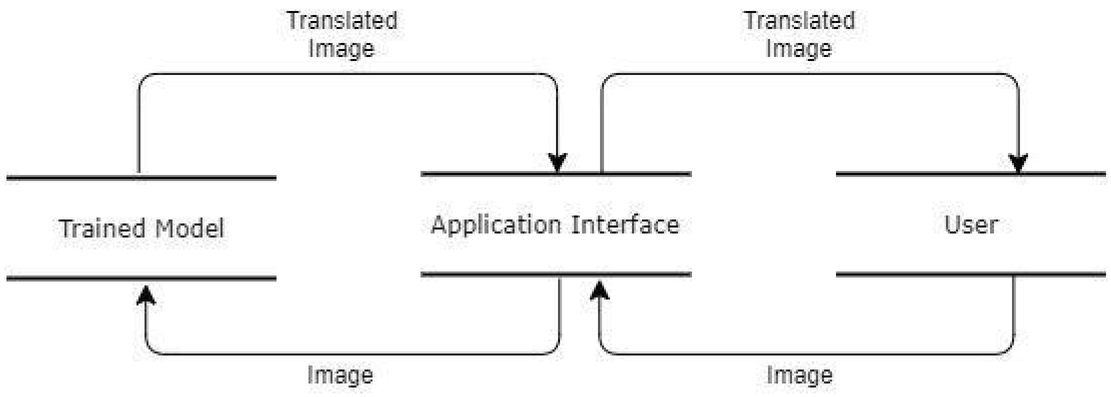
In a paired dataset, every image, say imgA, is manually mapped to some image, say imgB, in target domain, such that they share various features. Features that can be used to map an image (imgA/imgB) to its correspondingly mapped counterpart (imgB/imgA). Basically, pairing is done to make input and output share some common features. This mapping defines meaningful transformation of an image from one domain to another domain. So, when we have paired dataset, generator must take an input, say inputA, from domain DA and map this image to an output image, say genB, which must be close to its mapped counterpart. But we don't have this luxury in unpaired dataset, there is no pre-defined meaningful transformation that we can learn, so, we will create it. We need to make sure that there is some meaningful relation between input image and generated image. So, authors tried to enforce this by saying that Generator will map input image (inputA) from domain DA to some image in target domain DB, but to make sure that there is meaningful relation between these images, they must share some feature, features that can be used to map this output image back to input image, so there must be another generator that must be able to map back this output image back to original input. So, you can see this condition defining a meaningful mapping between inputA and genB.

In a nutshell, the model works by taking an input image from domain DA which is fed to our first generator GeneratorA→B whose job is to transform a given image from domain DA to an image in target domain DB. This new generated image is then fed to another generator GeneratorB→A which converts it back into an image, CyclicA, from our original domain DA.

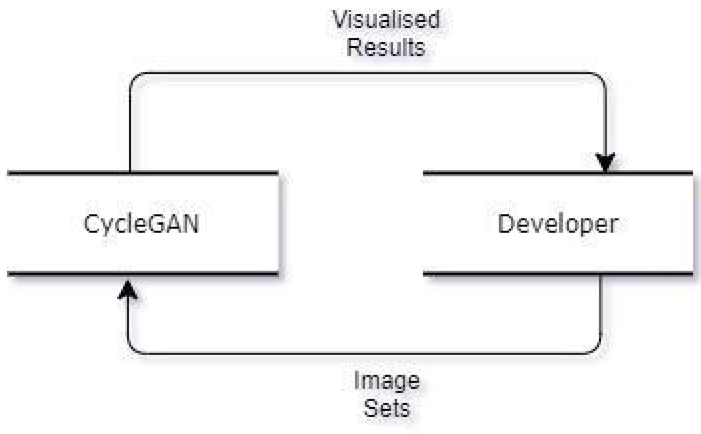
4.4 Sequence Diagram



4.5 Level 0 Data Flow Diagram for Trained System



4.6 Level 0 Data Flow Diagram for Training Phase



5.CODING

5.1 **Implementation of each package**:

train.py is a general-purpose training script. It works for various models (with option --model: e.g., pix2pix, cyclegan, colorization) and different datasets (with option --dataset_mode: e.g., aligned, unaligned, single, colorization). See the main README and training/test tips for more details.

test.py is a general-purpose test script. Once you have trained your model with train.py, you can use this script to test the model. It will load a saved model from --checkpoints_dir and save the results to --results_dir. See the main README and training/test tips for more details.

5.2 **File Implementations:**

__init__.py implements the interface between this package and training and test scripts. train.py and test.py call from data import create_dataset and dataset = create_dataset(opt) to create a dataset given the option opt.

base_dataset.py implements an abstract base class (ABC) for datasets. It also includes common transformation functions (e.g., get_transform, __scale_width), which can be later used in subclasses.

image_folder.py implements an image folder class. We modify the official PyTorch image folder code so that this class can load images from both the current directory and its subdirectories.

template_dataset.py provides a dataset template with detailed documentation. Check out this file if you plan to implement your own dataset.

aligned_dataset.py includes a dataset class that can load image pairs. It assumes a single image directory /path/to/data/train, which contains image pairs in the form of {A,B}. See here on how to prepare aligned datasets. During test time, you need to prepare a directory /path/to/data/test as test data.

unaligned_dataset.py includes a dataset class that can load unaligned/unpaired datasets. It assumes that two directories to host training images from domain A /path/to/data/trainA and from domain B /path/to/data/trainB respectively. Then you can train the model with the dataset flag --dataroot /path/to/data. Similarly, you need to prepare two directories /path/to/data/testA and /path/to/data/testB during test time.

single_dataset.py includes a dataset class that can load a set of single images specified by the path --dataroot /path/to/data. It can be used for generating CycleGAN results only for one side with the model option -model test.

util directory includes a miscellaneous collection of useful helper functions.

__init__.py is required to make Python treat the directory util as containing packages, get_data.py provides a Python script for downloading CycleGAN and pix2pix datasets. Alternatively, You can also use bash scripts such as download_pix2pix_model.sh and download_cyclegan_model.sh.
html.py implements a module that saves images into a single HTML file. It consists of functions such as add_header (add a text header to the HTML file), add_images (add a row of images to the HTML file), save (save the HTML to the disk). It is based on Python library dominate, a Python library for creating and manipulating HTML documents using a DOM API.
image_pool.py implements an image buffer that stores previously generated images. This buffer enables us to update discriminators using a history of generated images rather than the ones produced by the latest generators. The original idea was discussed in this paper. The size of the buffer is controlled by the flag --pool_size.
visualizer.py includes several functions that can display/save images and print/save logging information. It uses a Python library visdom for display and a Python library dominate (wrapped in HTML) for creating HTML files with images.
util.py consists of simple helper functions such as tensor2im (convert a tensor array to a numpy image array), diagnose_network (calculate and print the mean of average absolute value of gradients), and mkdirs (create multiple directories).

5.3 Training Code:

```
if __name__ == '__main__':
    opt = TrainOptions().parse()      # get training options
    dataset = create_dataset(opt)    # create a dataset given opt.dataset_mode and other options
    dataset_size = len(dataset)        # get the number of images in the dataset.
    print('The number of training images = %d' % dataset_size)
    model = create_model(opt)            # create a model given opt.model and other options
```

```
model.setup(opt)                    # regular setup: load and print networks; create schedulers

  visualizer = Visualizer(opt)     # create a visualizer that display/save images and plots
      total_iters = 0                    # the total number of training iterations

      for epoch in range(opt.epoch_count, opt.n_epochs + opt.n_epochs_decay + 1):      # outer loop
for      different epochs; we save the model by <epoch_count>, <epoch_count>+<save_latest_freq>
            epoch_start_time = time.time()    # timer for entire epoch
            iter_data_time = time.time()       # timer for data loading per iteration
            epoch_iter = 0                          # the number of training iterations in current epoch,
reset to 0 every          epoch
            visualizer.reset()                    # reset the visualizer: make sure it saves the results to
HTML at least           once every epoch

            for i, data in enumerate(dataset):    # inner loop within one epoch
                  iter_start_time = time.time()    # timer for computation per iteration
                  if total_iters % opt.print_freq == 0:
                        t_data = iter_start_time - iter_data_time

                  total_iters += opt.batch_size
                  epoch_iter += opt.batch_size
                  model.set_input(data)             # unpack data from dataset and apply preprocessing
                  model.optimize_parameters()     # calculate loss functions, get gradients, update
network          weights

                  if total_iters % opt.display_freq == 0:     # display images on visdom and save images
to a      HTML file
                        save_result = total_iters % opt.update_html_freq == 0
                        model.compute_visuals()
                        visualizer.display_current_results(model.get_current_visuals(), epoch,
save_result)
                if total_iters % opt.print_freq == 0:      # print training losses and save logging
information to           the disk
```

```
   losses = model.get_current_losses()



  t_comp = (time.time() - iter_start_time) / opt.batch_size
                    visualizer.print_current_losses(epoch, epoch_iter, losses, t_comp, t_data)
                    if opt.display_id > 0:
                        visualizer.plot_current_losses(epoch, float(epoch_iter) / dataset_size, losses)

                if total_iters % opt.save_latest_freq == 0:    # cache our latest model every
<save_latest_freq>   iterations
                    print('saving the latest model (epoch %d, total_iters %d)' % (epoch, total_iters))
                    save_suffix = 'iter_%d' % total_iters if opt.save_by_iter else 'latest'
                    model.save_networks(save_suffix)

                iter_data_time = time.time()
            if epoch % opt.save_epoch_freq == 0:              # cache our model every
<save_epoch_freq>   epochs
                print('saving the model at the end of epoch %d, iters %d' % (epoch, total_iters))
                model.save_networks('latest')
                model.save_networks(epoch)

            print('End of epoch %d / %d \t Time Taken: %d sec' % (epoch, opt.n_epochs +
          opt.n_epochs_decay, time.time() - epoch_start_time))
                 model.update_learning_rate()                     # update learning rates at the
end of every epoch.
```

5.4 Testing Code:

```
import os
from options.test_options import TestOptions
from data import create_dataset
from models import create_model
from util.visualizer import save_images
from util import html
if __name__ == '__main__':
    opt = TestOptions().parse()  # get test options
    # hard-code some parameters for test
    opt.num_threads = 0   # test code only supports num_threads = 1
    opt.batch_size = 1    # test code only supports batch_size = 1
    opt.serial_batches = True  # disable data shuffling; comment this line if results on randomly
      chosen images are needed.
    opt.no_flip = True    # no flip; comment this line if results on flipped images are needed.
    opt.display_id = -1   # no visdom display; the test code saves the results to a HTML file.
    dataset = create_dataset(opt)  # create a dataset given opt.dataset_mode and other options
    model = create_model(opt)      # create a model given opt.model and other options
    model.setup(opt)               # regular setup: load and print networks; create schedulers
    # create a website
    web_dir = os.path.join(opt.results_dir, opt.name, '{}_{}'.format(opt.phase, opt.epoch))  #
define  the website directory
    if opt.load_iter > 0:  # load_iter is 0 by default
        web_dir = '{:s}_iter{:d}'.format(web_dir, opt.load_iter)
    print('creating web directory', web_dir)
    webpage = html.HTML(web_dir, 'Experiment = %s, Phase = %s, Epoch = %s' % (opt.name,
      opt.phase, opt.epoch))
    # test with eval mode. This only affects layers like batchnorm and dropout.
    # For [pix2pix]: we use batchnorm and dropout in the original pix2pix. You can experiment it
with   and without eval() mode.
    # For [CycleGAN]: It should not affect CycleGAN as CycleGAN uses instancenorm without
      dropout.
    if opt.eval:
        model.eval()
    for i, data in enumerate(dataset):
        if i >= opt.num_test:  # only apply our model to opt.num_test images.
            break
            model.set_input(data)    # unpack data from data loader
            model.test()                    # run inference
            visuals = model.get_current_visuals()    # get image results




   img_path = model.get_image_paths()          # get image paths
            if i % 5 == 0:    # save images to an HTML file

      print('processing (%04d)-th image... %s' % (i, img_path))
            save_images(webpage, visuals, img_path, aspect_ratio=opt.aspect_ratio,
         width=opt.display_winsize)
       webpage.save()    # save the HTML
```

5.5 Cycle Gan Model Code:

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


class CycleGANModel(BaseModel):
        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)

            return parser
        def __init__(self, opt):
            """Initialize the CycleGAN class.



           Parameters:
                opt (Option class)-- stores all the experiment flags; needs to be a subclass of
                BaseOptions
            """
            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.load_networks>.
                  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
                      self.criterionGAN = networks.GANLoss(opt.gan_mode).to(self.device)    #
define GAN loss.
                      self.criterionCycle = torch.nn.L1Loss()
                      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=(opt.beta1, 0.999))
                        self.optimizer_D = torch.optim.Adam(itertools.chain(self.netD_A.parameters(),
self.netD_B.parameters()), lr=opt.lr, betas=(opt.beta1, 0.999))
                        self.optimizers.append(self.optimizer_G)
                        self.optimizers.append(self.optimizer_D)
        def set_input(self, input):
                """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.
                """
                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):
                """Run forward pass; called by both functions <optimize_parameters> and
<test>."""
                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):
                """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.
        """
        # 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):
        """Calculate GAN loss for discriminator D_A"""
        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):
        """Calculate GAN loss for discriminator D_B"""
        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):
        """Calculate the loss for generators G_A and G_B"""
        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()

          def optimize_parameters(self):
              """Calculate losses, gradients, and update network weights; called in every training
iteration"""
```

```
        # 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
```

6. TESTING

For any software that is newly developed, primary importance is given to the testing of the system. It goes to the developer to detect and correct errors in the software. Testing is the process by which the programmer will generate a set of test data, which gives a maximum probability of finding all types of errors that can occur in the software. Once source code has been generated, the software must be tested to uncover and correct as many errors as possible before it becomes operational. For this, we use a series of test cases that are designed using software testing techniques. These techniques provide systematic guidance for designing tests. The testing objectives are as follows:

- Testing is a process of executing with the intent of finding errors
- A good test case is that which has a high probability of finding a yet undiscovered error.
- A successful test is that which uncovers an as yet undiscovered error.

Our testing cases focused on training the CycleGAN with various options. However, unavailability of powerful hardware for the training process made this phase difficult. Our software worked without errors in all our trials with various datasets and with different training options.
The level of testing that our software has gone through: -

- Unit Testing
- System Testing

6.1 Unit Testing:-

Unit testing focuses on verification efforts on the smallest unit of software design in the module. This is also known as "Module Testing". The modules are tested separately. In this project, we have performed unit testing on the individual modules.

In our project, the modules were tested by providing dummy inputs to the various modules (train.py, test.py, etc) and testing whether it can read and write to the proper directory structures. The testing went successfully with no errors.

6.2 System Testing: -

System testing is the stage of implementation, which is aimed at ensuring that the system works accurately and efficiently before live operation commences. Testing is vital to the success of the system.

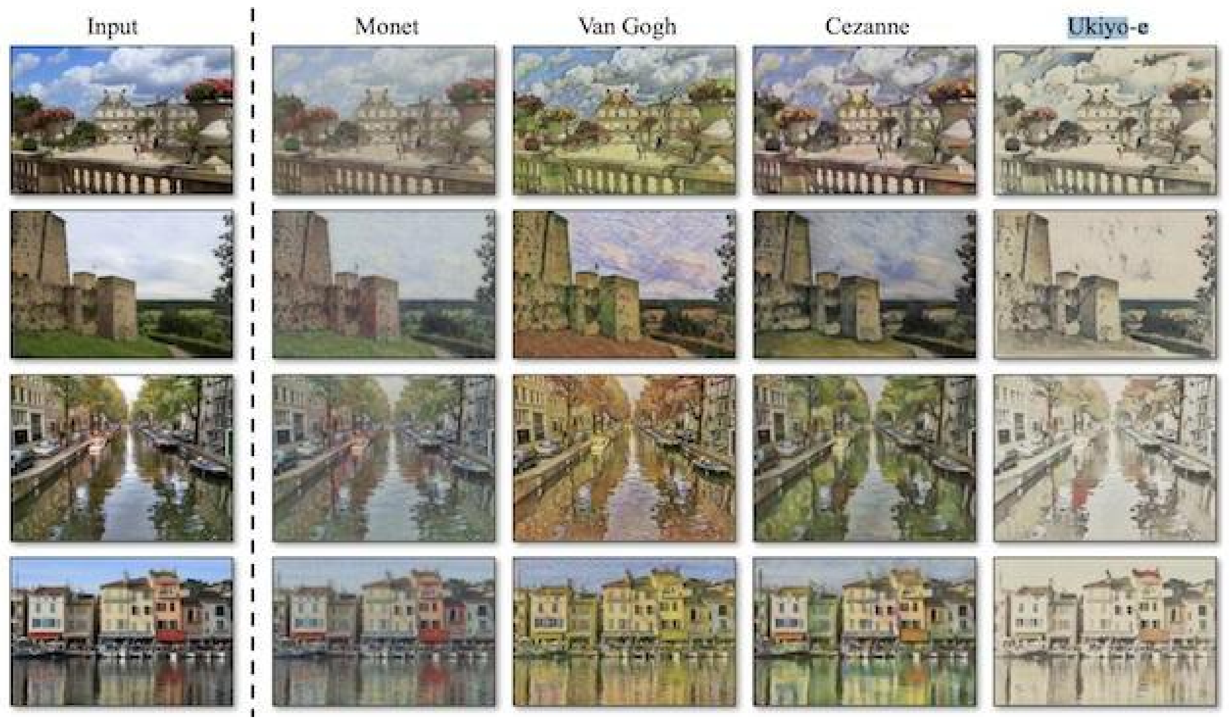
System testing makes a logical assumption that if all the parts of the system are correct, the goal will be successfully achieved. The candidate system is subject to a variety of tests online response. Volume, stress, recovery, and security and usability tests. We have performed system testing after our project was completed.

In our project, the system testing is done by running the entirety of code and seeing whether the training data is being read and properly fed through the neural network model, and if the results are being generated and stored properly. There were no errors in the program

7. USE CASES

1. Style Transfer

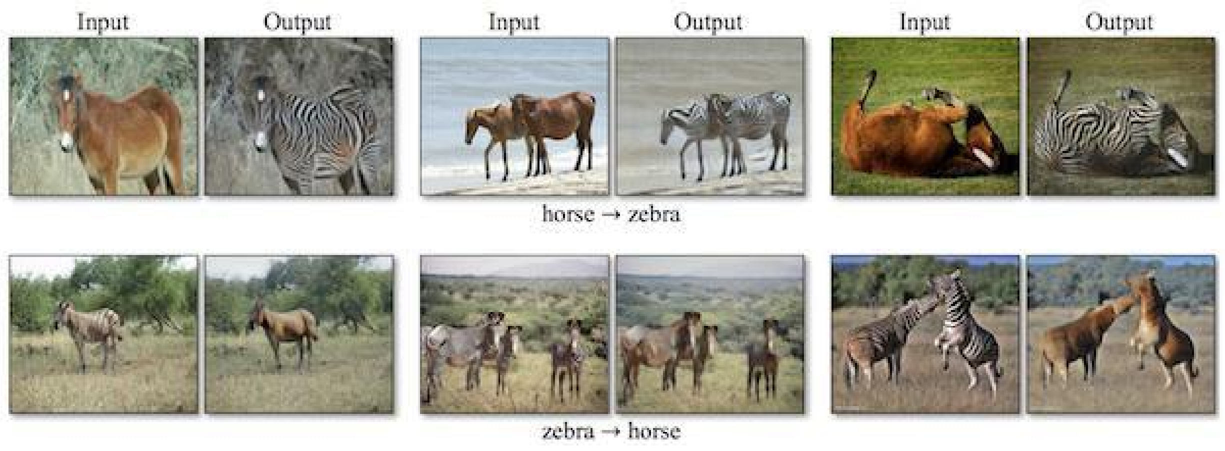
Style transfer refers to the learning of artistic style from one domain, often paintings, and applying the artistic style to another domain, such as photographs.
The CycleGAN is demonstrated by applying the artistic style from Monet, Van Gogh, Cezanne, and Ukiyo-e to photographs of landscapes.



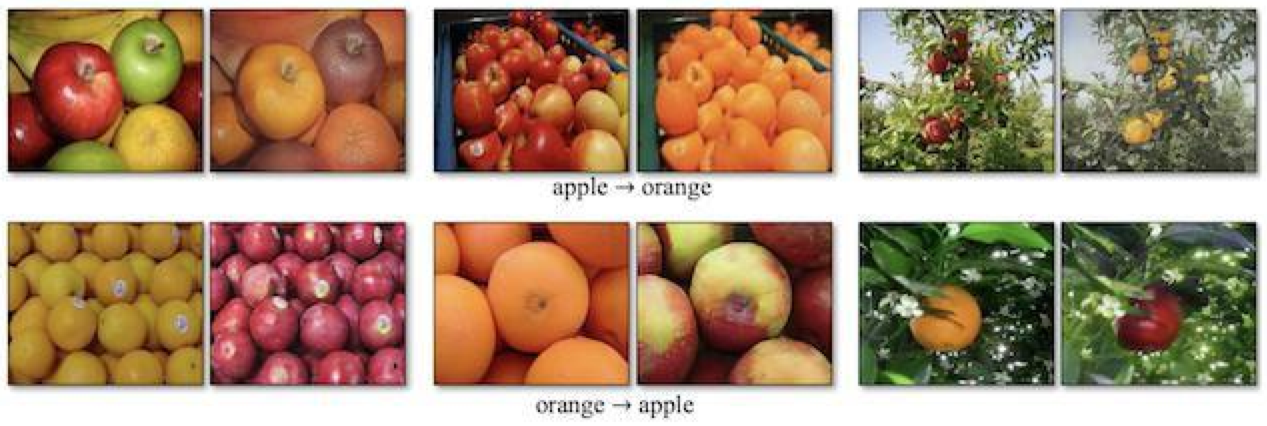
Example of Style Transfer from Famous Painters to Photographs of Landscapes

2. Object Transfiguration

Object transfiguration refers to the transformation of objects from one class, such as dogs into another class of objects, such as cats. The CycleGAN is demonstrated transforming photographs of horses into zebras and the reverse: photographs of zebras into horses. This type of transfiguration makes sense given that both horse and zebras look similar in size and structure, except for their coloring.



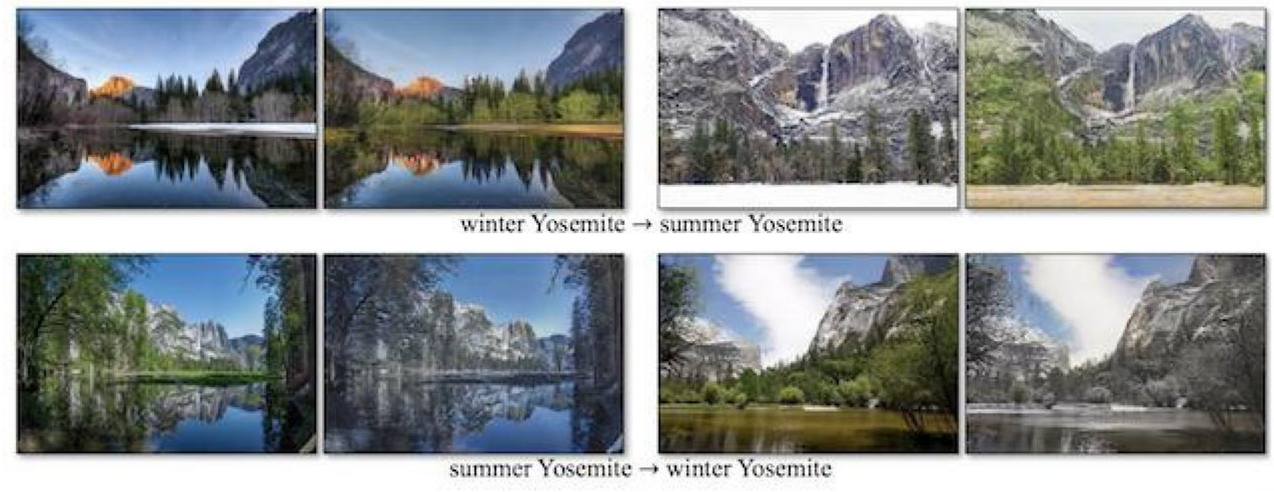
The CycleGAN is also demonstrated on translating photographs of apples to oranges, as well as the reverse: photographs of oranges to apples.
Again, this transfiguration makes sense as both oranges and apples have the same structure and size.



3. Season Transfer

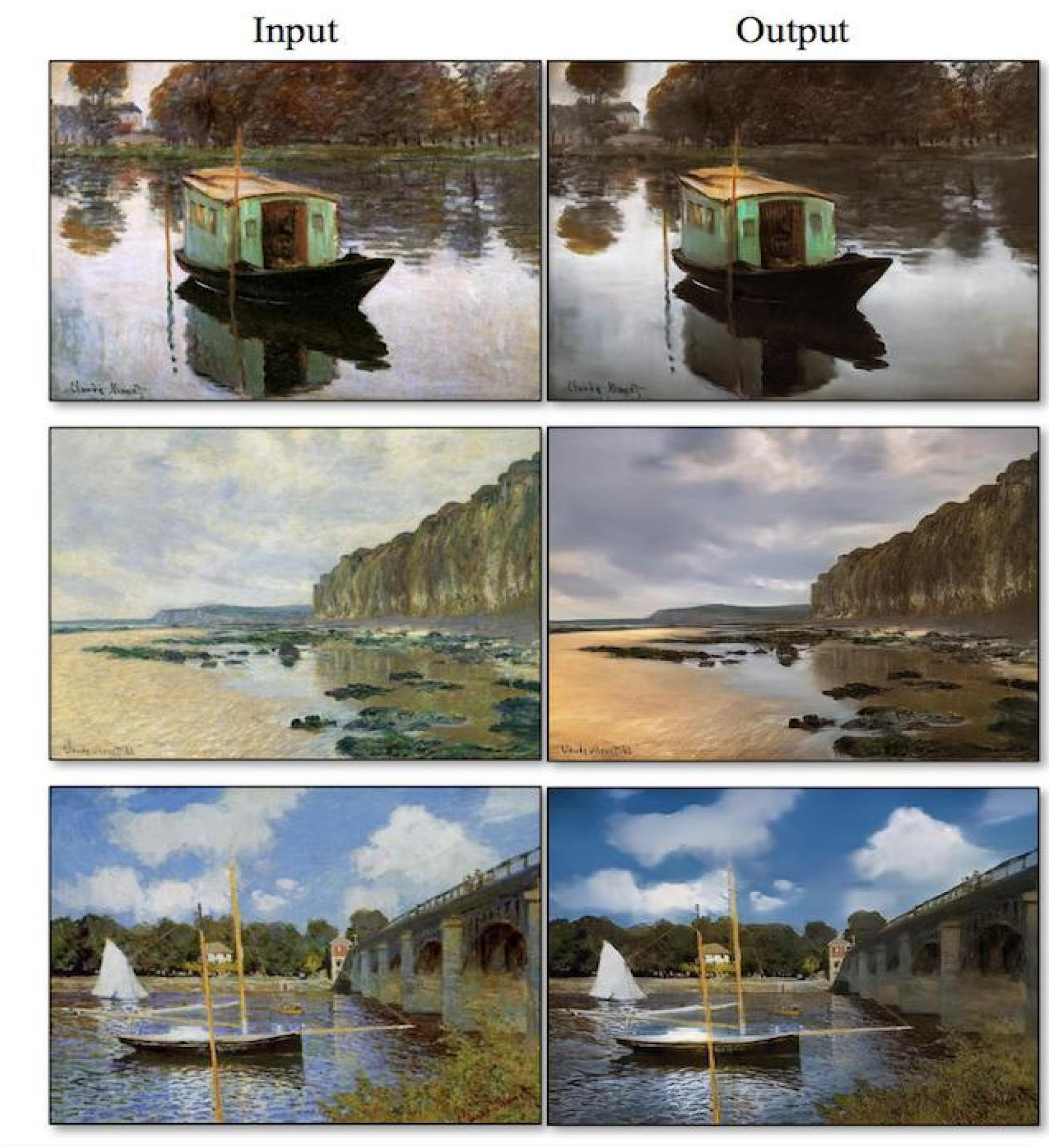
Season transfer refers to the translation of photographs taken in one season, such as summer, to another season, such as winter.

The CycleGAN is demonstrated on translating photographs of winter landscapes to summer landscapes, and the reverse of summer landscapes to winter landscapes.


winter Yosemite → summer Yosemite
summer Yosemite → winter Yosemite

4. Photograph Generation From Paintings

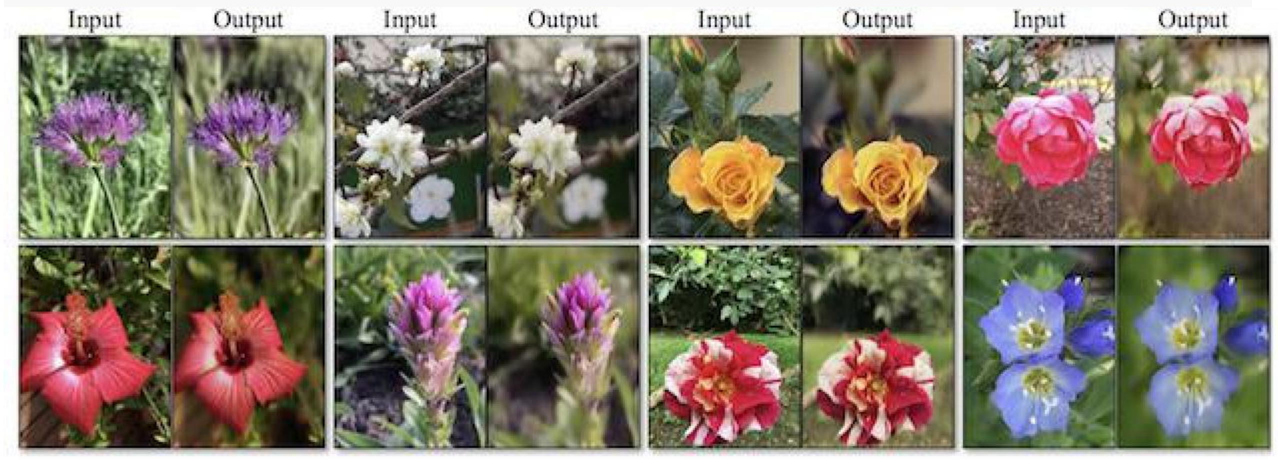
Photograph generation from paintings, as its name suggests, is the synthesis of photorealistic images given a painting, typically by a famous artist or famous scene.The CycleGAN is demonstrated on translating many paintings by Monet to plausible photographs.



5. Photograph Enhancement

Photograph enhancement refers to transforms that improve the original image in some way.

The CycleGAN is demonstrated on photo enhancement by improving the depth of field (e.g. giving a macro effect) on close-up photographs of flowers.



Example of Photograph Enhancement Improving the Depth of Field on Photos of Flowers

ADVANTAGES

Advantages of using a CycleGAN architecture is that there is no need for input-output pairs for training. The architecture automatically learns to map features from one domain to another. Since a major roadblock to Machine Learning projects is getting useful datasets, we have eliminated that roadblock by making the types of data sets that can be used, chosen from a vast array of unlabeled sets.

APPLICATIONS

Application of our project mainly lies in Computer Vision field. CV problems have always made use of neural network models. Solving CV problems have implications in many upcoming fields such as driverless cars, humanoid robots, novel CGI effects in movies etc. Studying CV field may also help us understand how our own visual system works.

Style transfer can also be used for data augmentation, in which small datasets are augmented to make a larger data set. Style transfer transfers style but keeps the semantics of an image, and therefore can be used to augment an image dataset with more varied images with the same semantic structure.

However style transfer isn’t limited to just image domains. Recent research has shown it to be useful when applied to other data sets such as vectors in a physical simulation, so style transfer is expected to have many more exciting applications in the future.

Style transfer can also be posed as a domain adaptation problem in the image space.

Image domain adaptation methods learn a mapping function to transform images in a source domain to have similar appearance to images in a target domain.

CycleGAN has been demonstrated on a range of applications including season translation, object transfiguration, style transfer, and generating photos from paintings.

8. IMPLEMENTATION

The proposed model can be straightforwardly implemented using modern machine learning frameworks such as Tensorflow, Keras, Torch, etc. For our purposes, we found that PyTorch (the python port of Torch framework) was the best suited and easiest to implement and test.

Style Transfer between 2 domains of Images, using a special but simple GAN architecture to perform our task. GAN on the generally needs a domain of images to train on and is consequently able in our case to capture the style of the image in its entirety.

The generator model starts with best practices for generators using the deep convolutional GAN, which is implemented using multiple residual blocks. The discriminator models use PatchGAN.

PatchGANs are used in the discriminator models to classify 70×70 overlapping patches of input images as belonging to the domain or having been generated. The discriminator output is then taken as the average of the prediction for each patch. The adversarial loss is implemented using a least-squared loss function.

For training, available local hardware wasn't powerful enough for the training to be feasible in reasonable time. Thankfully, google provides a cloud service called google colaboratory that became useful to us. Google colaboratory (also called colab) is a virtual environment running on Google's cloud servers. It allows us to run interactive python jupyter notebooks in the cloud environment. Jupyter notebooks are .ipynb files that allows us to run code interactively in a client-server setup. It can contain both text and python code snippets. The results of the training (the saved neural models and resulting image for each epoch of training) can be saved permanently to our Google Drive accounts, by mounting the drive to the colab virtual machines. The notebook we made can be viewed and opened here:
https://github.com/azharjuman/pytorch-CycleGAN-and-pix2pix/blob/master/Copy_of_CycleGAN.ipynb
You can use the notebook on your own Google account to train your own datasets.
The code of the notebook is provided below:

```
{
  "nbformat": 4,
  "nbformat_minor": 0,
  "metadata": {
    "colab": {
      "name": "Copy of CycleGAN",
      "provenance": [],
      "collapsed_sections": [],
      "include_colab_link": true
    },
    "kernelspec": {
      "name": "python3",
      "display_name": "Python 3"
    },
    "accelerator": "GPU"
  },
  "cells": [
    {
      "cell_type": "markdown",
      "metadata": {
        "id": "view-in-github",
        "colab_type": "text"
      },
      "source": [
        "<a href=\"https://colab.research.google.com/github/azharjuman/pytorch-CycleGAN-and-
pix2pix/blob/master/Copy_of_CycleGAN.ipynb\" target=\"_parent\"><img
src=\"https://colab.research.google.com/assets/colab-badge.svg\" alt=\"Open In Colab\"/></a>"
      ]
    },
    {
      "cell_type": "markdown",
      "metadata": {
        "id": "7wNjDKdQy35h",
        "colab_type": "text"
      },
      "source": [
        "# Install"
      ]
    },
    {
      "cell_type": "markdown",
      "metadata": {
        "id": "28dDUfrc8TAE",
        "colab_type": "text"
      },
      "source": [
        " ⭑ Before running anything, click 'Runtime' on menu bar, then click 'change runtime type'.
Make sure Hardware Accelerator is set to GPU."
      ]
    },
    {
      "cell_type": "markdown",
      "metadata": {
        "id": "0aVfE-9Q5gHu",
        "colab_type": "text"
      },
      "source": [
        " ⭑ Run these three code blocks once. (Click on each block, and then click the ▶  button on
the left side of the selected block)"
      ]
    },
    {
      "cell_type": "code",
      "metadata": {
        "id": "TRm-USlsHgEV",
        "colab_type": "code",
        "colab": {}
      },
      "source": [
        "!git clone https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix"
      ],
      "execution_count": null,
      "outputs": []
    },
    {
      "cell_type": "code",
      "metadata": {
        "id": "Pt3igws3eiVp",
        "colab_type": "code",
        "colab": {}
      },
      "source": [
        "import os\n",
        "os.chdir('pytorch-CycleGAN-and-pix2pix/')"
      ],
      "execution_count": null,
      "outputs": []
    },
    {
      "cell_type": "code",
      "metadata": {
        "id": "z1EySlOXwwoa",
        "colab_type": "code",
        "colab": {}
      },
   "source": [
     "!pip install -r requirements.txt"
   ],
   "execution_count": null,
   "outputs": []
 },
 {
   "cell_type": "markdown",
   "metadata": {
     "id": "dHsaCLhO44Kl",
     "colab_type": "text"
   },
   "source": [
     " ⭑ Mounting your google drive  . Run the block, it will show you link for authorisation. Go to the link location, then allow access, copy the code that pops up, paste it at "Enter your authorization code:""
   ]
 },
 {
   "cell_type": "code",
   "metadata": {
     "id": "r7Khp0eD4vJ7",
     "colab_type": "code",
     "colab": {}
   },
   "source": [
     "from google.colab import drive \n",
     "drive.mount('./gdrive') "
   ],
   "execution_count": null,
   "outputs": []
 },
    {
      "cell_type": "markdown",
      "metadata": {
        "id": "8daqlgVhw29P",
        "colab_type": "text"
      },
      "source": [
        "# Datasets\n",
        "\n",
        "Download one of the official datasets with:\n",
        "\n",
        "-       `bash ./datasets/download_cyclegan_dataset.sh [apple2orange, orange2apple,
summer2winter_yosemite, winter2summer_yosemite, horse2zebra, zebra2horse, monet2photo,
style_monet, style_cezanne, style_ukiyoe, style_vangogh, sat2map, map2sat, cityscapes_photo2label,
cityscapes_label2photo, facades_photo2label, facades_label2photo, iphone2dslr_flower]`\n",
        "\n",
        "Or use your own dataset by creating the appropriate folders and adding in the images.\n",
        "\n",
        "-    Create a dataset folder under `/dataset` for your dataset.\n",
        "-    Create subfolders `testA`, `testB`, `trainA`, and `trainB` under your dataset's folder.
Place any images you want to transform from a to b (cat2dog) in the `testA` folder, images you want
to transform from b to a (dog2cat) in the `testB` folder, and do the same for the `trainA` and `trainB`
folders."
      ]
    },
    {
      "cell_type": "markdown",
      "metadata": {
        "id": "wEWr00PGJns3",
        "colab_type": "text"
      },
```

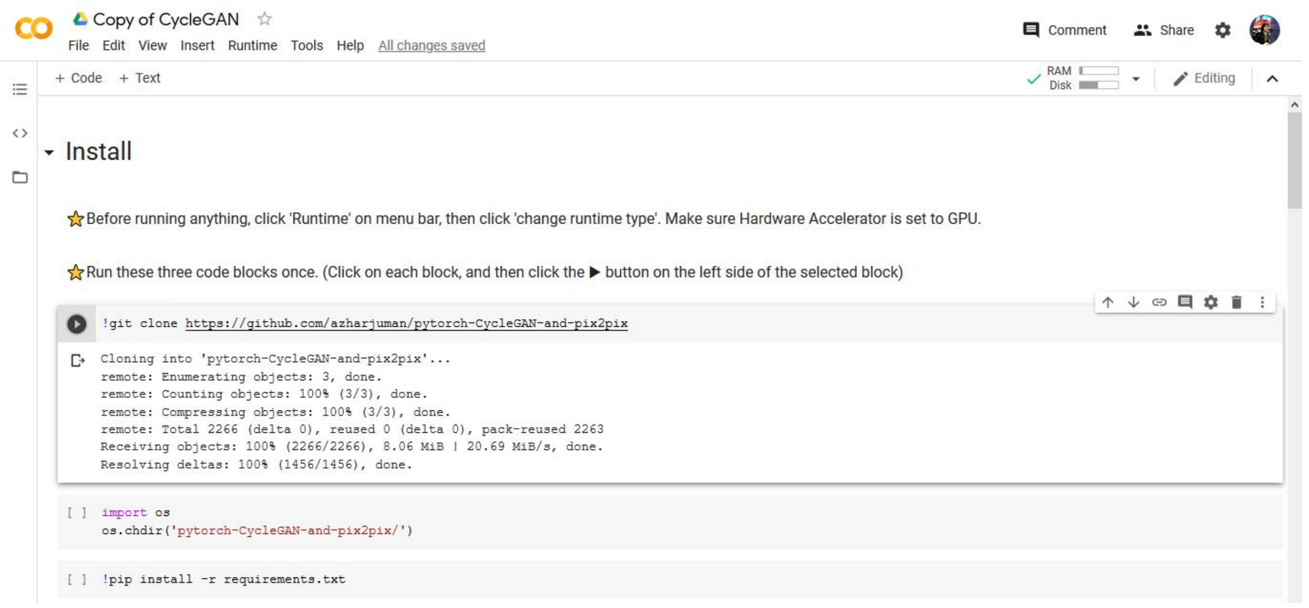
"source": [
"⭐ Run the block below to download your dataset . change 'horse2zebra' with the dataset you are training."
]
},
{
"cell_type": "code",
"metadata": {
"id": "vrdOettJxaCc",
"colab_type": "code",
"colab": {}
},
"source": [
"!bash ./datasets/download_cyclegan_dataset.sh horse2zebra"
],
"execution_count": null,
"outputs": []
},
{
"cell_type": "markdown",
"metadata": {
"id": "yFw1kDQBx3LN",
"colab_type": "text"
},
"source": [
"# Training\n",
"\n",
"-   `python train.py --dataroot ./datasets/horse2zebra --name horse2zebra --model cycle_gan`\n",
"\n",
"Change the `--dataroot` and `--name` to your own dataset's path and model's name. Use `--gpu_ids 0,1,..` to train on multiple GPUs and `--batch_size` to change the batch size. I've found that a batch size of 16 fits onto 4 V100s and can finish training an epoch in ~90s.\n",

```
            "\n",
            "Once your model has trained, copy over the last checkpoint to a format that the testing model can automatically detect:\n",
            "\n",
            "Use `cp ./checkpoints/horse2zebra/latest_net_G_A.pth ./checkpoints/horse2zebra/latest_net_G.pth` if you want to transform images from class A to class B and `cp ./checkpoints/horse2zebra/latest_net_G_B.pth ./checkpoints/horse2zebra/latest_net_G.pth` if you want to transform images from class B to class A.\n"
        ]
    },
    {
        "cell_type": "markdown",
        "metadata": {
            "id": "XMeq2YLTKxqE",
            "colab_type": "text"
        },
        "source": [
            " ⭑   During training, the program will periodically save checkpoints to your google drive in \"My Drive/cyclegan/horse2zebra/checkpoints/\" folder.\n",
            "Replace horse2zebra in all the paths with the name of the dataset you're training. Run the block below    to start training."
        ]
    },
    {
        "cell_type": "code",
        "metadata": {
            "id": "0sp7TCT2x9dB",
            "colab_type": "code",
            "colab": {}
        },
        "source": [
        "!python train.py --dataroot ./datasets/horse2zebra --name horse2zebra --model cycle_gan --checkpoints_dir \"./gdrive/My Drive/cyclegan/horse2zebra/checkpoints\" --n_epochs 50 --n_epochs_decay 0 --save_epoch_freq 2"
      ],
      "execution_count": null,
      "outputs": []
    },
    {
      "cell_type": "markdown",
      "metadata": {
        "id": "i20YKuSHM-B7",
        "colab_type": "text"
      },
      "source": [
        " ⭑ You should keep the computer open for this one. Google will automatically terminate the session if left uninteracted for 90 minutes, so either interact with the notebook hourly, or follow one of the solutions here:\n",
        "https://stackoverflow.com/questions/61254168/prevent-a-google-colab-process-from-being-disconnected"
      ]
    },
    {
      "cell_type": "markdown",
      "metadata": {
        "id": "opa7-9DTOzXC",
        "colab_type": "text"
      },
      "source": [
        " ⭑ If by chance your session ended/disconnected, You will have to run all the blocks in the 'Install' section (cloning git, mounting google drive etc.), download the dataset from the 'Datasets' section, and then run the block below to continue training  (remember to replace the horse2zebra with your dataset name everywhere). Also insert the line ' --epoch_count x ' into the below code
```

```
block, replacing x with your latest saved epoch number"
   ]
  },
  {
   "cell_type": "code",
   "metadata": {
    "id": "3PobMz-JVIy5",
    "colab_type": "code",
    "colab": {}
   },
   "source": [
    "!python train.py --dataroot ./datasets/horse2zebra --name horse2zebra --model cycle_gan -
-checkpoints_dir \"./gdrive/My Drive/cyclegan/horse2zebra/checkpoints\" --n_epochs 50 --
n_epochs_decay 0 --save_epoch_freq 2 --continue_train"
   ],
   "execution_count": null,
   "outputs": []
  },
  {
   "cell_type": "markdown",
   "metadata": {
    "id": "9UkcaFZiyASl",
    "colab_type": "text"
   },
   "source": [
    "# Testing\n",
    "\n",
    "-    `python test.py --dataroot datasets/horse2zebra/testA --name horse2zebra_pretrained -
-model test --no_dropout`\n",
    "\n",
    "Change the `--dataroot` and `--name` to be consistent with your trained model's
configuration.\n",
        "\n",
        "> from https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix:\n",
        "> The option --model test is used for generating results of CycleGAN only for one side. This option will automatically set --dataset_mode single, which only loads the images from one set. On the contrary, using --model cycle_gan requires loading and generating results in both directions, which is sometimes unnecessary. The results will be saved at ./results/. Use --results_dir {directory_path_to_save_result} to specify the results directory.\n",
        "\n",
        "> For your own experiments, you might want to specify --netG, --norm, --no_dropout to match the generator architecture of the trained model."
      ]
    },
    {
      "cell_type": "code",
      "metadata": {
        "id": "uCsKkEq0yGh0",
        "colab_type": "code",
        "colab": {}
      },
      "source": [
        "!python test.py --dataroot datasets/horse2zebra/testA --name horse2zebra_pretrained --model test --no_dropout"
      ],
      "execution_count": null,
      "outputs": []
    },
    {
      "cell_type": "markdown",
      "metadata": {
        "id": "XoVVxsofCiQq",
        "colab_type": "text"
      },
            "source": [
                "⭐ The results of the test will be stored in your Google drive."
            ]
        }
    ]
}
```

The training and testing results is done by running the code snippets in the notebook in colab. Detailed instructions can be viewed from within the notebook.

Some screenshots of the colab implementation:



Install

⭐Before running anything, click 'Runtime' on menu bar, then click 'change runtime type'. Make sure Hardware Accelerator is set to GPU.

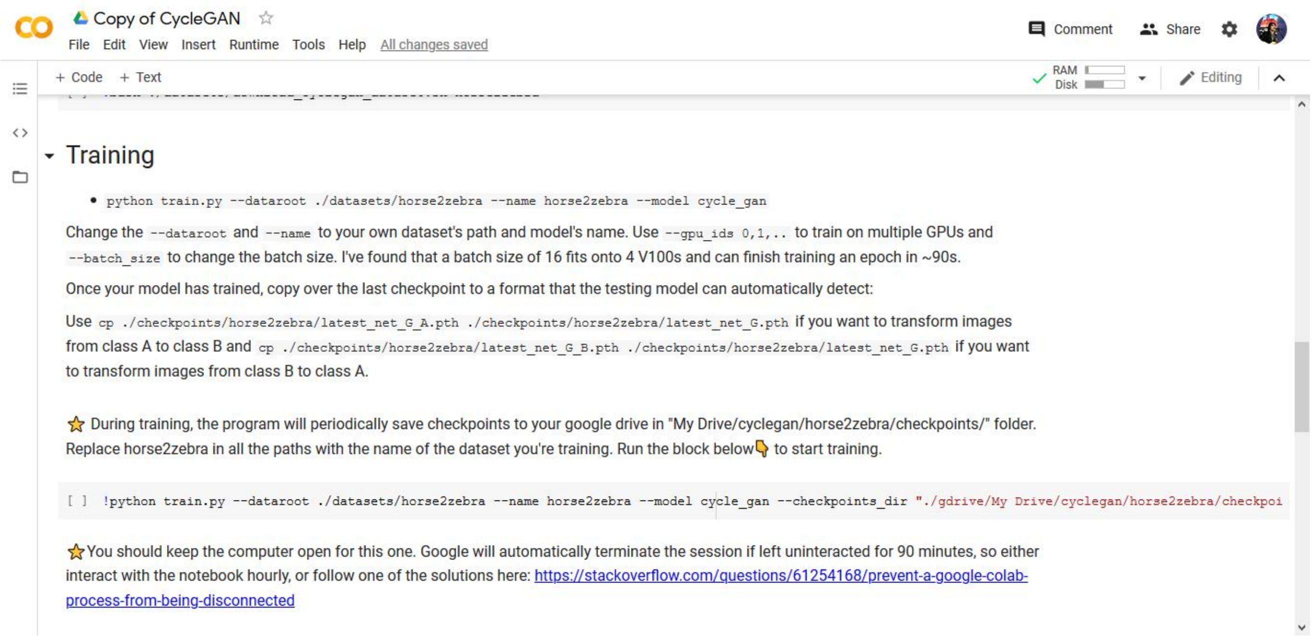
⭐Run these three code blocks once. (Click on each block, and then click the ▶ button on the left side of the selected block)

```
!git clone https://github.com/azharjuman/pytorch-CycleGAN-and-pix2pix
```

```
Cloning into 'pytorch-CycleGAN-and-pix2pix'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 2266 (delta 0), reused 0 (delta 0), pack-reused 2263
Receiving objects: 100% (2266/2266), 8.06 MiB | 20.69 MiB/s, done.
Resolving deltas: 100% (1456/1456), done.
```

```
import os
os.chdir('pytorch-CycleGAN-and-pix2pix/')
```

```
!pip install -r requirements.txt
```



Training

- `python train.py --dataroot ./datasets/horse2zebra --name horse2zebra --model cycle_gan`

Change the `--dataroot` and `--name` to your own dataset's path and model's name. Use `--gpu_ids 0,1,..` to train on multiple GPUs and `--batch_size` to change the batch size. I've found that a batch size of 16 fits onto 4 V100s and can finish training an epoch in ~90s.

Once your model has trained, copy over the last checkpoint to a format that the testing model can automatically detect:

Use `cp ./checkpoints/horse2zebra/latest_net_G_A.pth ./checkpoints/horse2zebra/latest_net_G.pth` if you want to transform images from class A to class B and `cp ./checkpoints/horse2zebra/latest_net_G_B.pth ./checkpoints/horse2zebra/latest_net_G.pth` if you want to transform images from class B to class A.

⭐ During training, the program will periodically save checkpoints to your google drive in "My Drive/cyclegan/horse2zebra/checkpoints/" folder. Replace horse2zebra in all the paths with the name of the dataset you're training. Run the block below👇 to start training.

```
[ ] !python train.py --dataroot ./datasets/horse2zebra --name horse2zebra --model cycle_gan --checkpoints_dir "./gdrive/My Drive/cyclegan/horse2zebra/checkpoi
```

⭐You should keep the computer open for this one. Google will automatically terminate the session if left uninteracted for 90 minutes, so either interact with the notebook hourly, or follow one of the solutions here: https://stackoverflow.com/questions/61254168/prevent-a-google-colab-process-from-being-disconnected

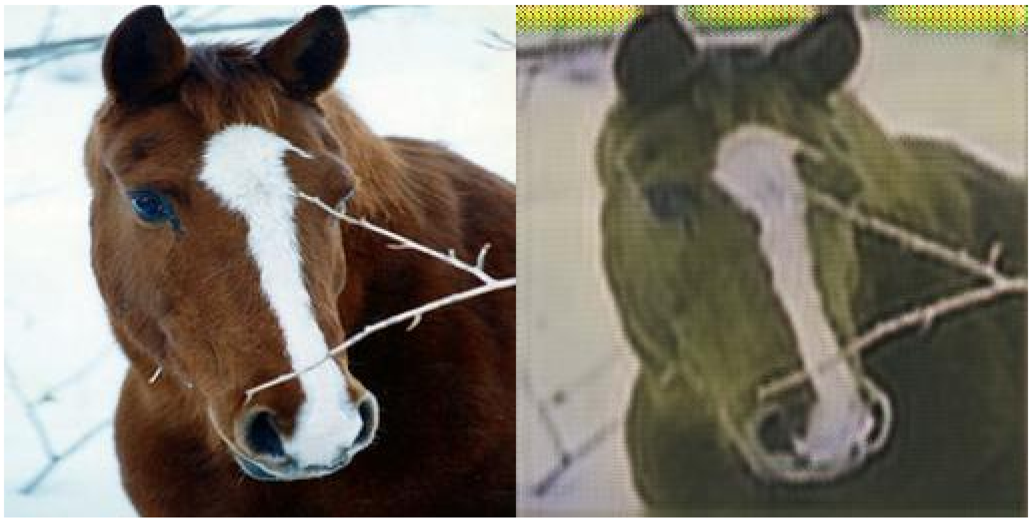
9. RESULTS

The results, even if low quality, were satisfactory given that we didn't have access to powerful hardware as much as we'd like. All our code ran perfectly without any errors, and we got trained models and resulting images generated out of it. The resulting images for different datasets at different epochs are given below. You can see the progress made by the system in these generated images. Each training set took more than 8 hours to train.

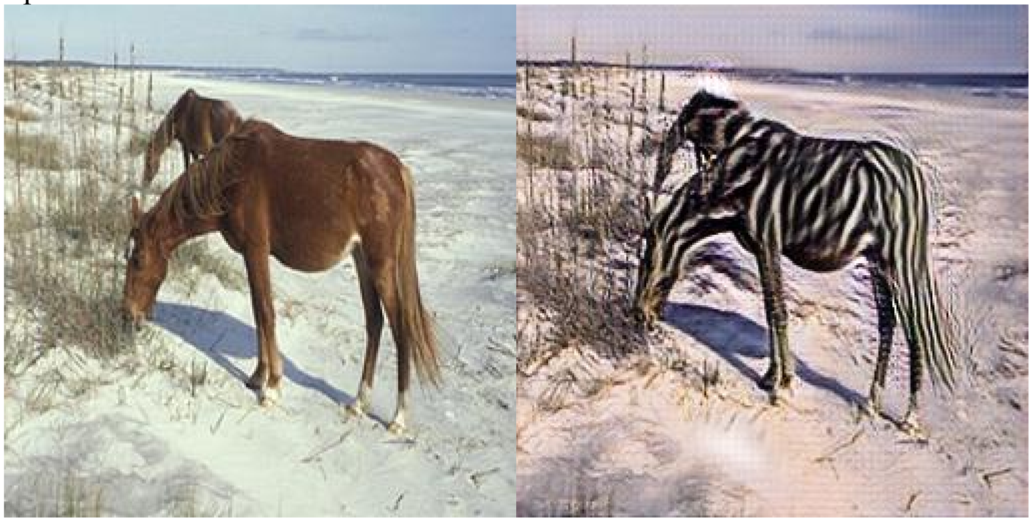
The horses2zebra dataset:

This dataset is used to train a CycleGAN that transfigures an image of a horse to an image of a zebra (or vice versa).

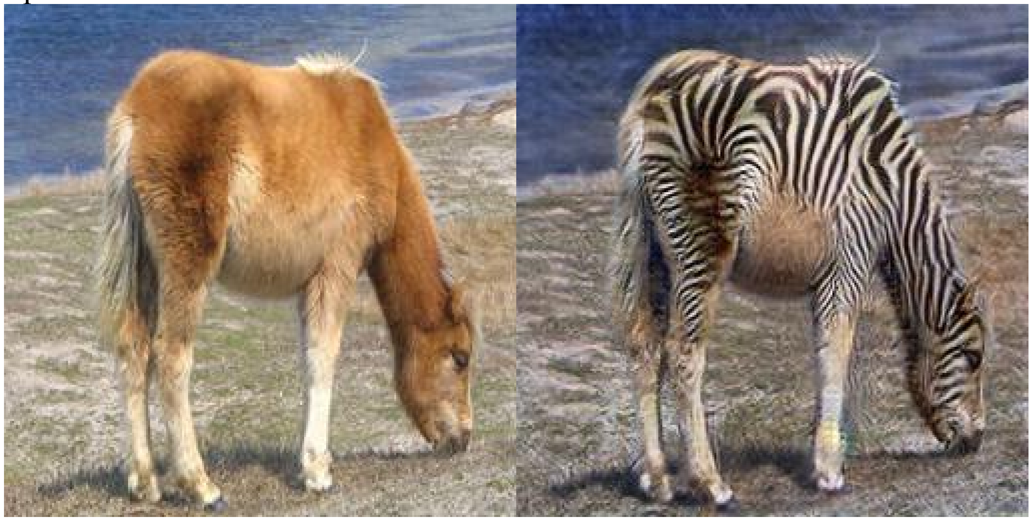
Epoch 1:



Epoch 22:



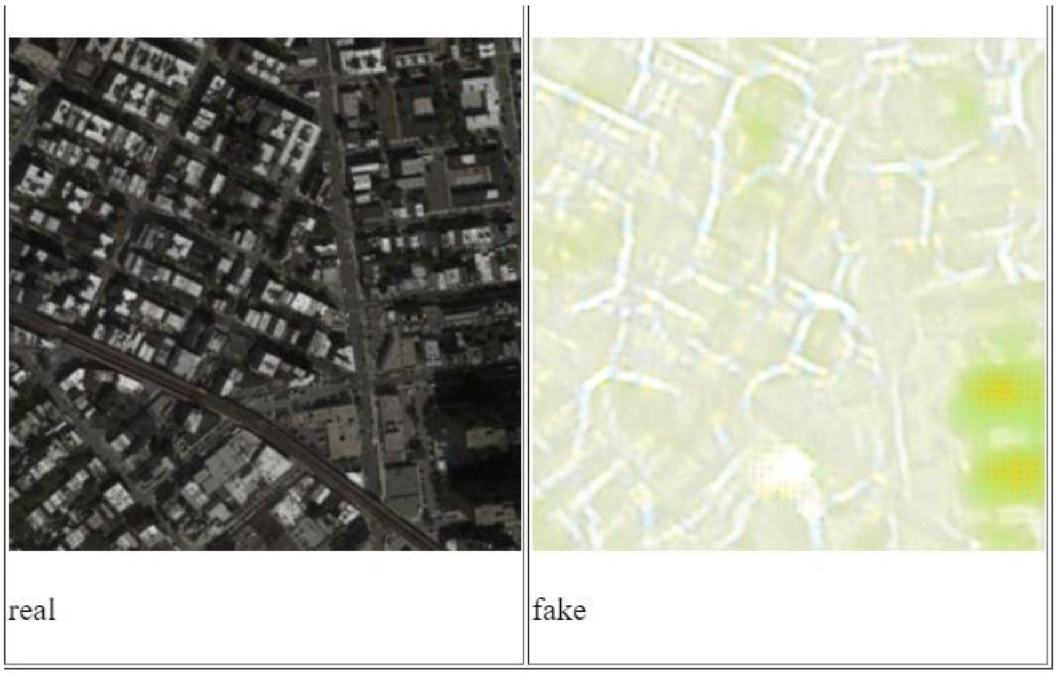
Epoch 46:



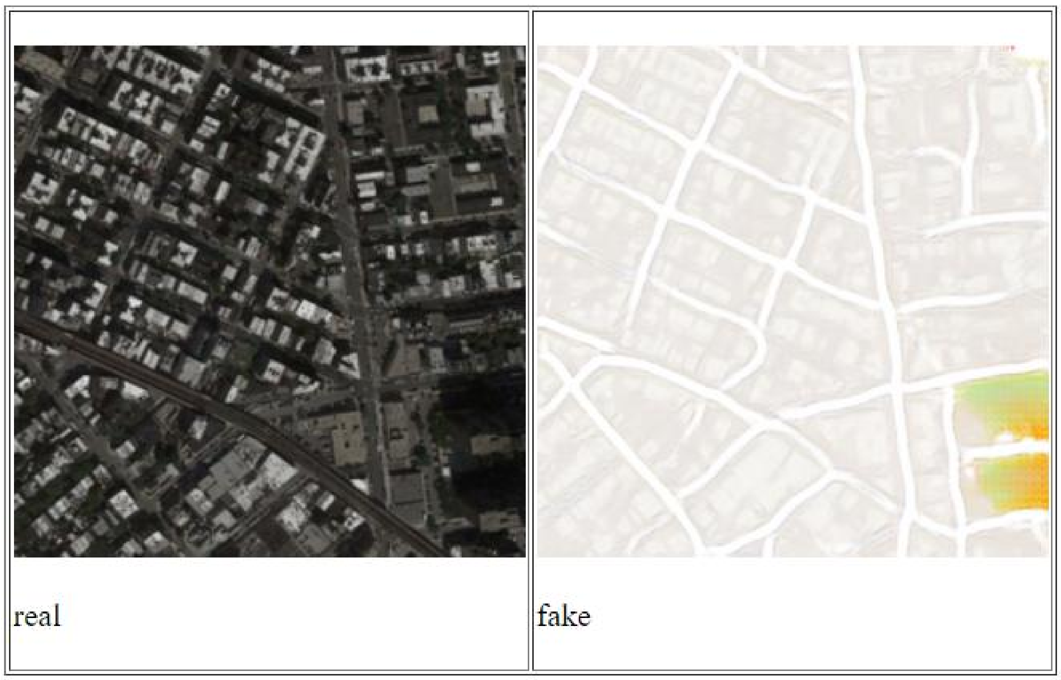
The sat2map dataset:

This dataset is used to train a CycleGAN that can translate aerial/satellite photos of a terrain into maps similar to the ones that Google Maps use.

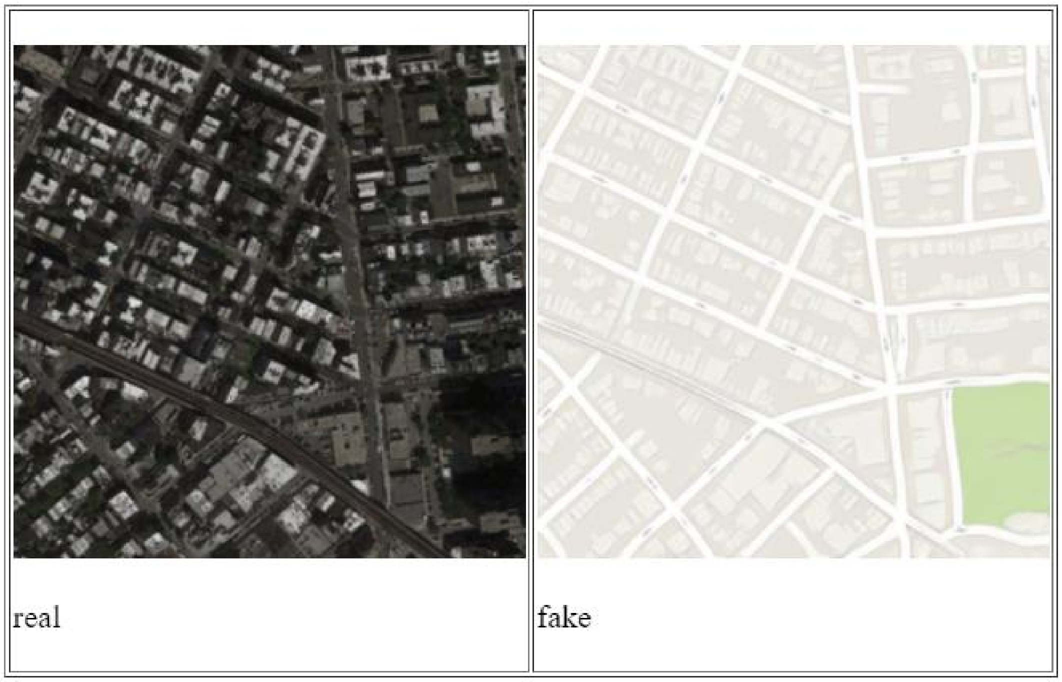
Epoch 1:



Epoch 11:



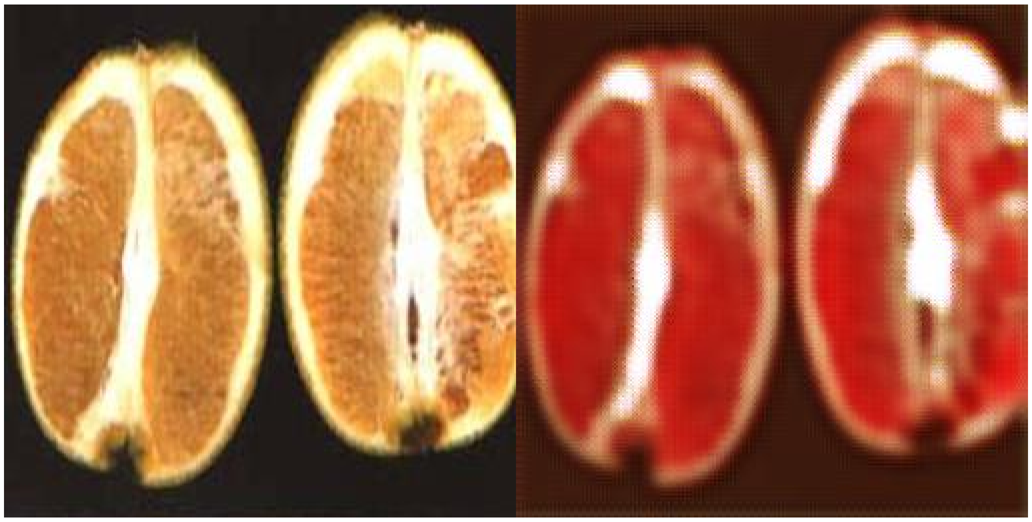
Epoch 50:



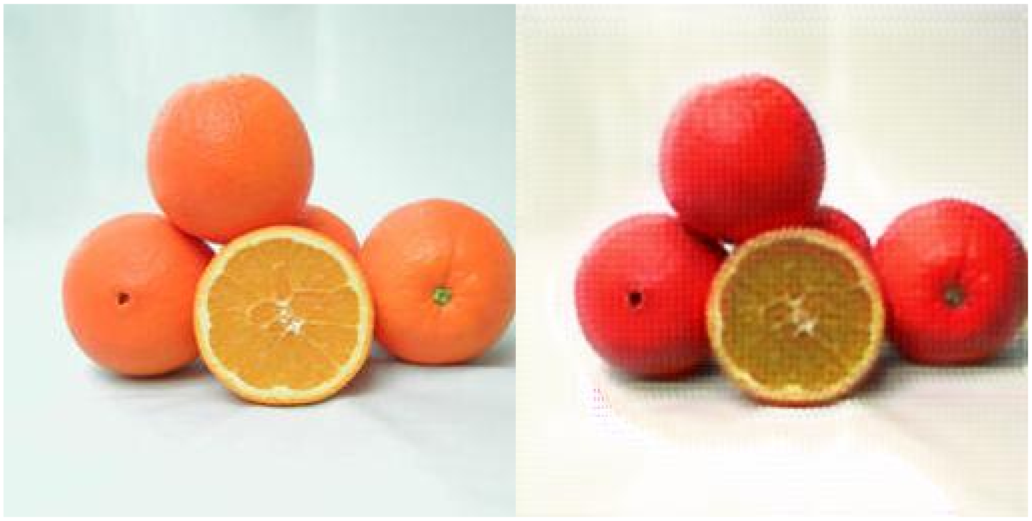
The orange2apple dataset:

This dataset is used to train a CycleGAN that can transfigure pictures of oranges to a picture of apples:

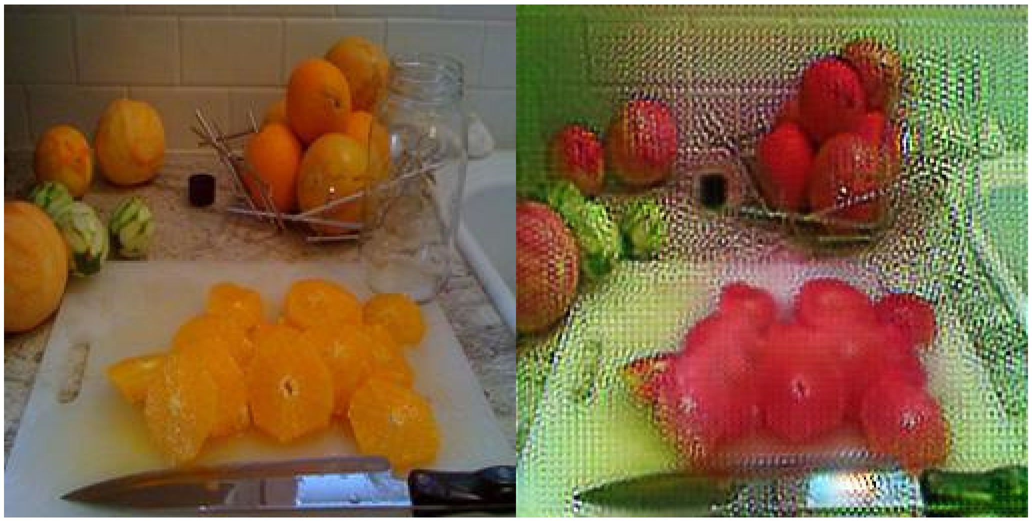
Epoch 1:



Epoch 23:



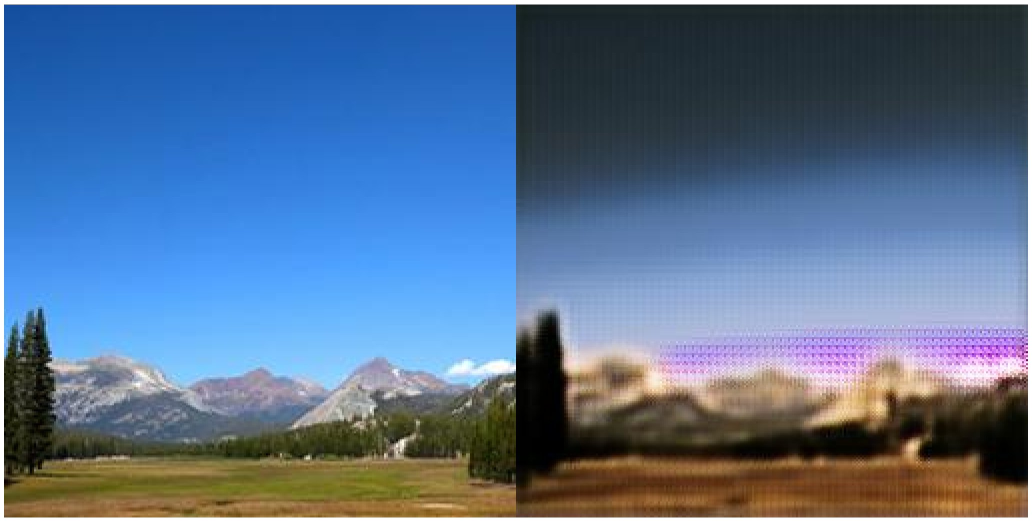
Epoch 50:



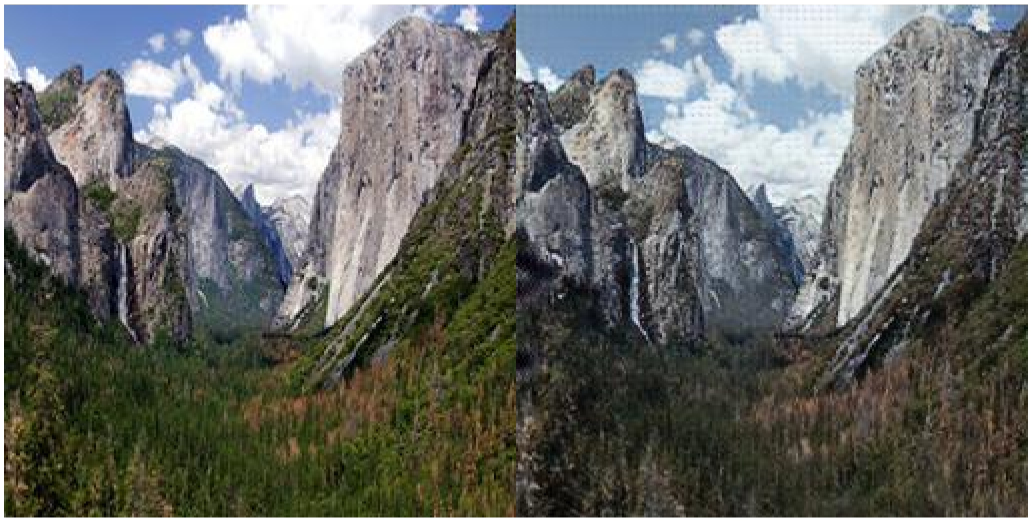
The summer2winter dataset:

This dataset is used to train a CycleGAN that can shift the tones and lighting in a winter landscape photo to that of summer.

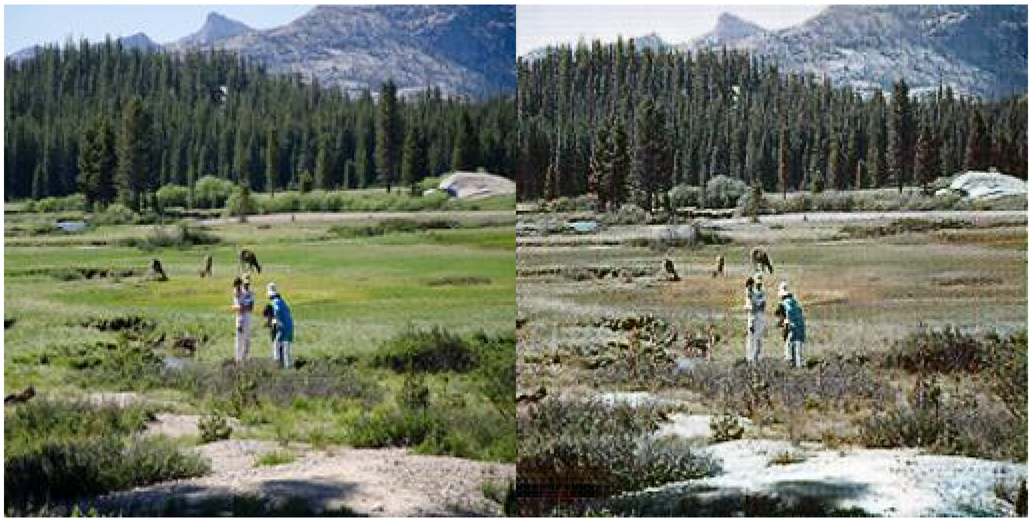
Epoch 1:



Epoch 25:



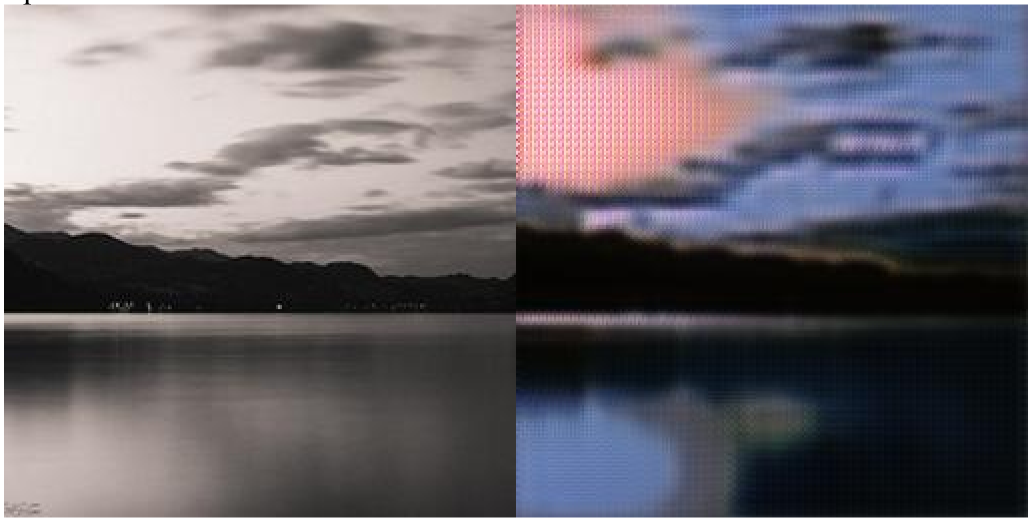
Epoch 33:



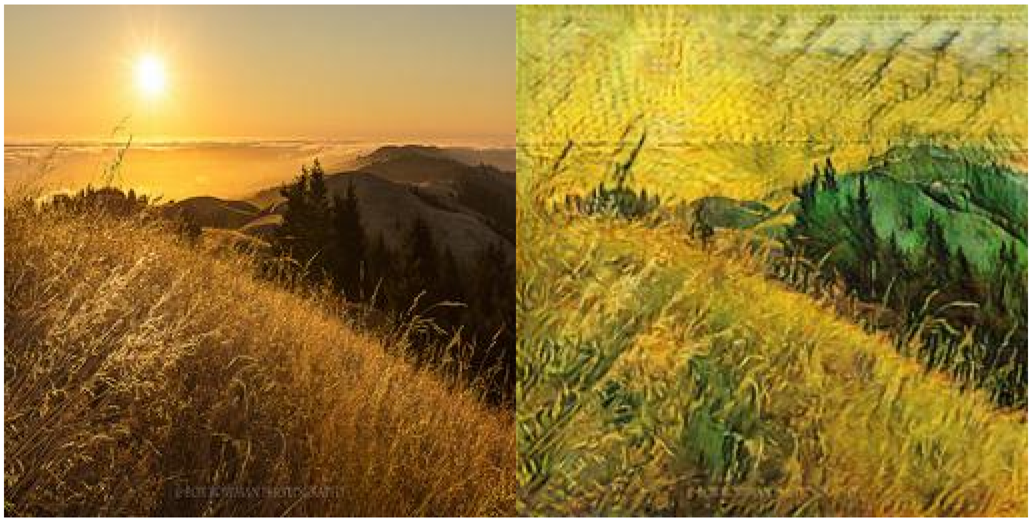
The stylevangogh dataset:

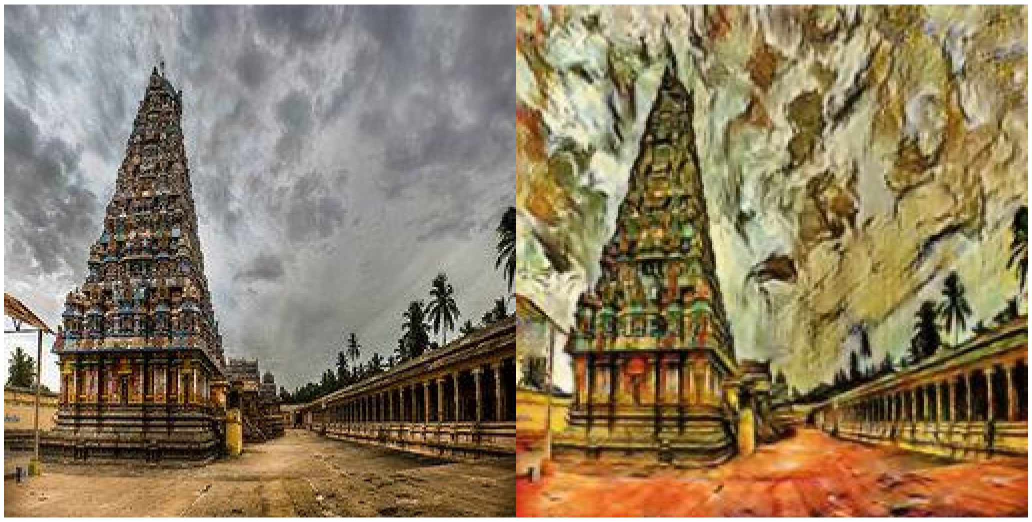
This dataset of photographs and Vangogh paintings is used to train a CycleGAN that can convert photos to the style of Vangogh paintings.

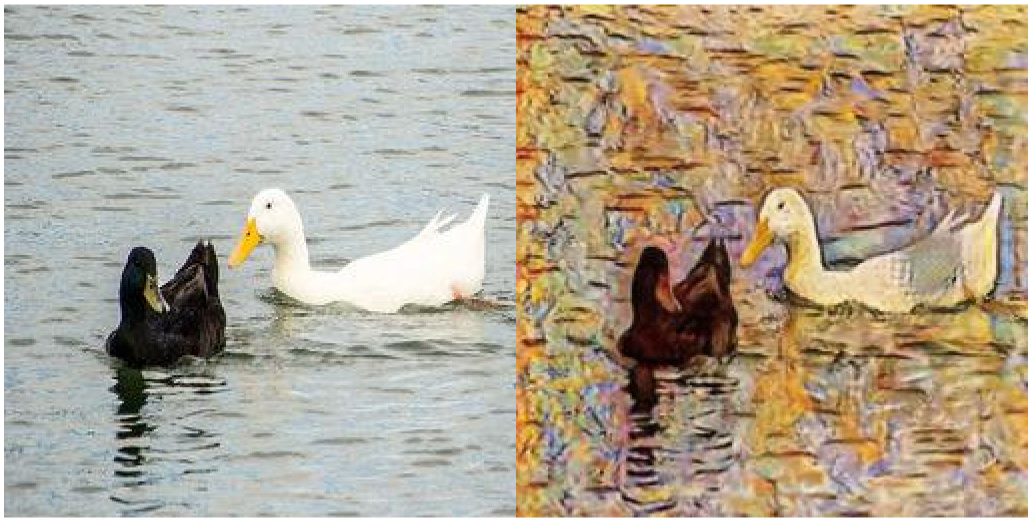
Epoch 5:



Epoch 50:







10. CONCLUSION

With the increasing popularity of applications of machine learning to various fields, style transfer provides a way to translate data through different domains. CycleGAN’s core concept, the idea of cycle consistency, is at the foundation of many developing fields in machine learning research that use unsupervised and unpaired data. Our project showed that even a bit of training with varied datasets can result in exciting possibilities. Therefore, the results of this research suggest even more, that artificial intelligence techniques may be successfully used to develop our visual science research and industrial fields even more.

11. FUTURE WORK

Through the work done in this project, we have shown that machine learning certainly does have the capacity to pick up on sometimes complex patterns that may be difficult for humans to program on. The next steps involved in this project come in three different aspects. The first of aspect that could be improved in this project is augmenting and increasing the size of the dataset. We feel that more data would be beneficial in ridding the model of any bias based on specific patterns in the source. There is also a question as to whether or not the size of our dataset is sufficient. Another concern is that of hardware. Due to limited access to performance hardware, our training time was limited. Our model could be vastly improve with better hardware that could run higher resolution models. Taking this a step further, image style transfer could be extended to video style transfer.

12. REFERENCES

1. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks by Jun-Yan-Zhu , Taesung Park, Philip Isola.
   Link: https://arxiv.org/abs/1703.10593

2. Dataset Link: https://people.eecs.berkeley.edu/~taesung_park/CycleGAN/datasets/

3. GAN Wiki : https://en.wikipedia.org/wiki/Generative_adversarial_network

4. Understanding and Implementing CycleGAN in TensorFlow By Hardik Bansal
   Link: https://hardikbansal.github.io/CycleGANBlog/

5. Pytorch Implementation: https://pytorch.org/docs/stable/torch.html