ILLINOIS INSTITUTE OF TECHNOLOGY
Department of Computer Science

CS512 Computer Vision

Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks

Naveen Raju Sreerama Raju Govinda Raju
nsreeramarajugovinda@hawk.iit.edu
A20516868

Jai Raj Choudhary
jchoudhary@hawk.iit.edu
A20517449

Prof. Gady Agam

Submission Date: 18th April 2023

Table of Contents

1. Abstract

Image-to-image translation is a type of computer vision problem where the goal is to learn how to transform an input image into an output image using a set of aligned image pairs that are used as training examples. To have paired training data available for the task, though, may not always be possible. Hence in the new method we learn the translation of images from a source domain X to a target domain Y, even in the absence of paired examples.

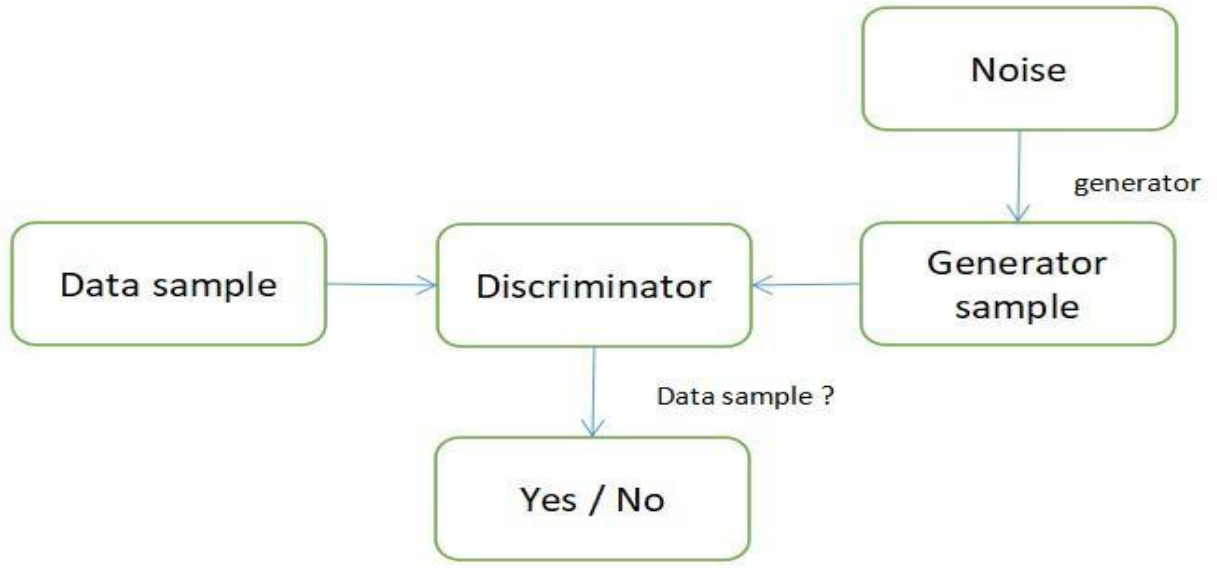
2. Problem statement

Unpaired image-to-image translation is carried out using Cycle-Consistent Adversarial Networks (CycleGAN), according to the method. Traditional approaches rely on paired data, which can be hard to get and keep up with. By creating a system that can do image-to-image translation without paired data and making use of the idea of cycle consistency, the approach seeks to get around this barrier. The methodology makes it more realistic and doable for real-world image translation applications by allowing the use of unaligned and unpaired data for training.

3. Introduction

Generative Adversarial Networks (GAN's)

Generative Adversarial Networks (GANs), were first developed in 2014 by academics at the University of Montreal under the direction of Ian Goodfellow (now at OpenAI). Two neural network models that are in competition make up the fundamental idea of GANs. One model, called the generator, creates samples using random noise as its input. The objective of the other model, known as the discriminator, which receives samples from both the generator and the actual training data, is to separate the two sources. In a constant game between these two networks, the generator strives to create increasingly realistic samples, and the discriminator strives to become better at telling actual data apart from fake.



4. Dataset

Horse and Zebra images are considered for our project.

Source1 : TensorFlow datasets.

Data set size:

A) Training Data: Horses: 267 images, Zebra: 334 images

B) Testing Data: Horses: 30 images, Zebra: 35 images

Source2: Kaggle CycleGAN datasets.

Data set size:

A) Training Data: Horses: 1037 images, Zebra: 1034 images

B)Testing Data: Horses: 120 images, Zebra: 140 images

5. Proposed solution:

The details of the architecture of the proposed solution are as follows:

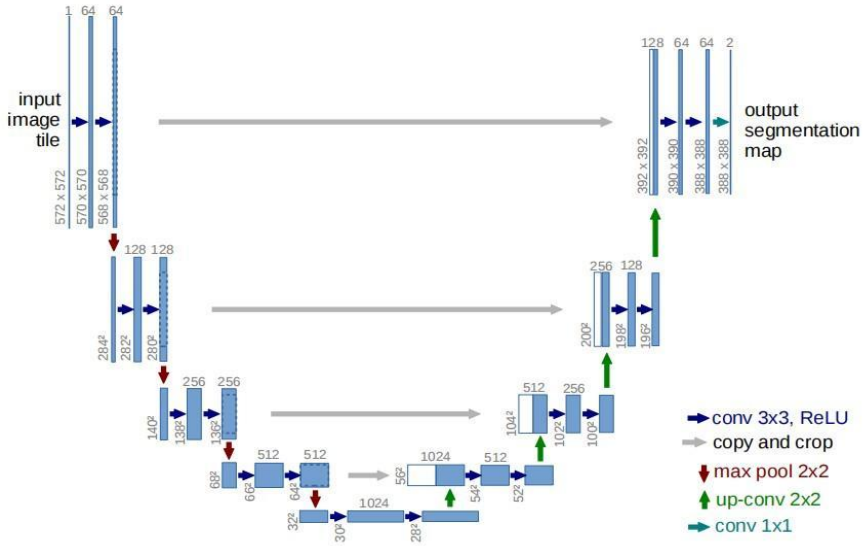
a. Generator :

Generator is based on U-Net. U-Net consists of encoder (down sampling) and decoder (up sampling). Encoder and decoder relate to skip connections.

In general, each block of encoder has Convolution->LeakyRelu->Instance normalization.

In general, each block of decoder has Transpose Convolution ->Relu->Concatenation->Convolution->Relu->Dropout.
At last Convolution ->Tanh->Output

The encoder uses convolutional layers to capture features from the input image, while the decoder uses transposed convolutional layers to upsample and reconstruct the image. Skip connections are used to connect the encoder and decoder at multiple levels, allowing for the retention of fine-grained details.



There two different custom designed generators used in this project: "custom_unet_generator_v2" and "custom_unet_generator_v3."

Generator 1 (custom_unet_generator_v2):

Takes an input of 256*256*3

Encoder: Consists of several blocks:

First block:

- 1st CNN layer

Number of filters = 64, Kernel size = 3, Activation = LeakyRelu, Bias = False, Stride = 1, padding = same

- 2nd CNN layer
  Number of filters = 64, Kernel size = 3, Activation = LeakyRelu, Bias = False, Stride = 2, padding = same

Second block:

- 1st CNN layer
  Number of filters = 128, Kernel size = 3, Activation = LeakyRelu, padding = same, Bias = False, Stride = 1

- 2nd CNN layer
  Number of filters = 128, Kernel size = 3, Activation = LeakyRelu, padding = same, Bias = False, Stride = 2, Normalization = Instance normalization

Third block:

- 1st CNN layer
  Number of filters = 256, Kernel size = 3, Activation = LeakyRelu, padding = same, Bias = False, Stride = 1

- 2nd CNN layer
  Number of filters = 256, Kernel size = 3, Activation = LeakyRelu, padding = same, Bias = False, Stride = 2, Normalization = Instance normalization

Fourth block:

- 1st CNN layer
  Number of filters = 512, Kernel size = 3, Activation = LeakyRelu, padding = same, Bias = False, Stride = 1

- 2nd CNN layer
  Number of filters = 512, Kernel size = 3, Activation = LeakyRelu, padding = same, Bias = False, Stride = 2, Normalization = Instance normalization

Bottleneck block:

- 1st CNN layer
  Number of filters = 1024, Kernel size = 3, Activation = LeakyRelu, padding = same, Bias = False, Stride = 2

- 2nd CNN layer
  Number of filters = 1024, Kernel size = 3, Activation = LeakyRelu, padding = same, Bias = False, Stride = 1

Decoder: Consists of several blocks

First block:

- 1st CNN layer
  Number of filters = 512 , Kernel size = 3, Activation = LeakyRelu, Bias = False, Stride = 2, padding = same
  Concatenation : Encoder fourth block

- 2nd CNN layer
  Number of filters = 512 , Kernel size = 3, Activation = LeakyRelu, Bias = False, Stride = 1, padding = same

- 3rd CNN layer
  Number of filters = 512 , Kernel size = 3, Activation = LeakyRelu, Bias = False, Stride = 2, padding = same
  Dropout = 0.5

Second block:

- 1st CNN layer
  Number of filters = 256, Kernel size = 3, Activation = LeakyRelu, Bias = False, Stride = 2, padding = same
  Concatenation : Encoder third block

- 2nd CNN layer

Number of filters = 256, Kernel size = 3, Activation = LeakyRelu, Bias = False, Stride = 1, padding = same

- 3rd CNN layer
  Number of filters = 256, Kernel size = 3, Activation = LeakyRelu, Bias = False, Stride = 2, padding = same
  Dropout = 0.5

Third block:

- 1st CNN layer
  Number of filters = 128, Kernel size = 3, Activation = LeakyRelu, Bias = False, Stride = 2, padding = same
  Concatenation : Encoder second block

- 2nd CNN layer
  Number of filters = 128, Kernel size = 3, Activation = LeakyRelu, Bias = False, Stride = 1, padding = same

- 3rd CNN layer
  Number of filters = 128, Kernel size = 3, Activation = LeakyRelu, Bias = False, Stride = 2, padding = same
  Dropout = 0.5

Fourth block:

- 1st CNN layer
  Number of filters = 64, Kernel size = 3, Activation = LeakyRelu, Bias = False, Stride = 2, padding = same
  Concatenation : Encoder first block

- 2nd CNN layer
  Number of filters = 64, Kernel size = 3, Activation = LeakyRelu, Bias = False, Stride = 1, padding = same

- 3rd CNN layer
  Number of filters = 64, Kernel size = 3, Activation = LeakyRelu, Bias = False, Stride = 2, padding = same
  Dropout = 0.5

- Transpose convolution
  Number of filters = 3, kernel size = 3, Stride = 2, padding = same, Activation = tanh

Generator 2 (custom_unet_generator_v3):

Takes an input of 256*256*3

Encoder and decoder have similar architecture as custom_unet_generator_v2

But the number of filters in Encoder is as follows: 40,80,160,320.
Bottle neck layer has 640 filters. In decoder number of filters is as follows 320,160,80,40.

b. Generator (Pytorch) :

The generator is composed of four main parts: an initial convolutional layer, a series of downsampling convolutional blocks, a set of residual blocks, and a series of upsampling convolutional blocks. The downsampling blocks reduce the spatial resolution of the input image while increasing the number of channels, and the upsampling blocks increase the spatial resolution while decreasing the number of channels. The residual blocks are used to capture the low-frequency details of the input image and help to prevent the model from losing important information during the downsampling and upsampling stages.

The initial convolutional layer applies a 7x7 convolution to the input image and applies instance normalization and ReLU activation. The downsampling blocks consist of two convolutional layers with stride 2, which reduce the spatial resolution by a factor of 2 and increase the number of channels by a factor of 2. The upsampling blocks consist of one transpose convolutional layer with stride 2, which doubles the spatial resolution, followed by a standard convolutional layer that reduces the number of channels by a factor of 2. Each residual block contains two convolutional layers with the same number of channels, followed by instance normalization and ReLU activation. Finally, the last layer applies a 7x7 convolution followed by a hyperbolic tangent activation function to produce the output image.

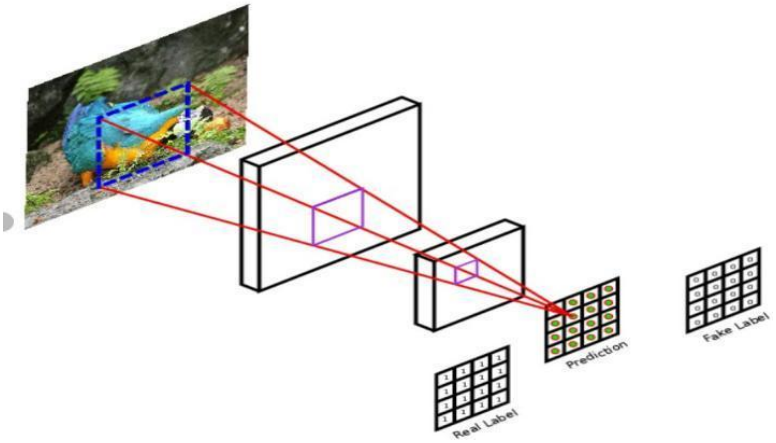
The Generator class takes as input the number of channels in the input and output images (img_channels) as well as two hyperparameters, num_features and num_residuals. The num_features parameter specifies the number of channels in the

initial convolutional layer, which is doubled in each downsampling block and halved in each upsampling block. The num_residuals parameter specifies the number of residual blocks to use in the generator. The test function creates a random input image of size 256x256 and passes it through the generator, printing the shape of the output image.

c. Discriminator :

Discriminator used is a modified PatchGAN classifier, the classifier tries to classify each patch of images if it is real or fake.

In general each block of discriminator has convolution->LeakyRelu->Instance normalization



There two different custom designed discriminators used in this project: “custom_unet_descriminator_v2” and “custom_unet_discriminator_dilated_v2”

Discriminator 1 (custom_unet_descriminator_v2):

Input of 256*256*3

- First Convolution layer:
  Number of filters = 32, Kernel size = 4, Activation = LeakyRelu, Bias = False, Strides = 1
- Second Convolution layer:
  Number of filters = 64, Kernel size = 4, Activation = LeakyRelu, Bias = False, Strides = 2
- Third Convolution layer:
  Number of filters = 128, Kernel size = 4, Activation = LeakyRelu, Bias = False, Strides = 2, Normalization = Instance normalization
- Fourth Convolution layer:
  Number of filters = 256, Kernel size = 4, Activation = LeakyRelu, Bias = False, Strides = 2, Normalization = Instance normalization, Zero padding
- Fifth Convolution layer:
  Number of filters = 512, Kernel size = 4, Activation = LeakyRelu, Bias = False, Strides = 1 , Normalization = Instance normalization, Zero padding
- Sixth Convolution layer:
  Number of filters = 512, Kernel size = 4,Activation = LeakyRelu, Bias = False, Strides = 1 , Normalization = Instance normalization
- Seventh convolution layer:
  Number of filters = 1, Kernel size = 4, Strides = 1

Discriminator 2 (custom_unet_discriminator_dilated_v2):

Input of 256*256*3

- First Convolution layer:
  Number of filters = 32, Kernel size = 4, Activation = LeakyRelu, Bias = False, Strides = 1

- Second Convolution layer:
  Number of filters = 64, Kernel size = 4, Activation = LeakyRelu, Bias = False, Strides = 2

- Third Convolution layer:
  Number of filters = 128, Kernel size = 4, Activation = LeakyRelu, Bias = False, Strides = 2, Normalization = Instance normalization

- Fourth layer Dilated Convolution:
  Number of filters = 256, Kernel size = 3, Bias = False, Strides = 1, Dilation rate = 2, Normalization = Instance normalization

- Fifth Convolution layer:
  Number of filters = 512, Kernel size = 4, Activation = LeakyRelu, Bias = False, Strides = 2 , Normalization = Instance normalization, Zero padding

- Sixth layer, Dilated Convolution:
  Number of filters = 1024, Kernel size = 3, Bias = False, Strides = 1, Dilation rate = 2, Zero padding

- Seventh convolution layer:
  Number of filters = 1, Kernel size = 4, Strides = 1

Discriminator 3 (pix2pix.discriminator):

Input of 256*256*3

- First Convolution layer:
  Number of filters = 64, Kernel size = 4, Activation = LeakyRelu, Bias = False, Strides = 2

- Second Convolution layer:
  Number of filters = 128, Kernel size = 4, Activation = LeakyRelu, Bias = False, Strides = 2

- Third Convolution layer:
  Number of filters = 256, Kernel size = 4, Activation = LeakyRelu, Bias = False, Strides = 2, Normalization = Batch normalization, Zero padding

- Fourth Convolution layer:
  Number of filters = 512, Kernel size = 4, Activation = LeakyRelu, Bias = False, Strides = 1, Normalization = Batch normalization, Zero padding

- Fifth convolution layer:
  Number of filters = 1, Kernel size = 4, Strides = 1

Here we have two main components of proposed approach which are generators and discriminators. We consider 2 domains here, Domain A -Horse and Domain B - Zebra. Since we want to make the model work in both direction that is A->B and B->A. Due to this we will have two generators, Generator G(A->B) and Generator F(B->A), and we have two discriminators, Discriminator Y(A) and Discriminator Y(B).

Let us consider Domain A images as X
Let us consider Domain B images as Y

d. Discriminator (Pytorch) :

This is an implementation of a discriminator network for a Generative Adversarial Network (GAN). The GAN consists of a generator and a discriminator network, and the goal is to train the generator to produce images that can deceive the discriminator into thinking they are real images. The discriminator network is responsible for distinguishing between real and fake images.

The discriminator network in this implementation takes an input image with 3 channels (RGB) of size 256x256 and produces a single scalar output that represents the probability of the input being a real image. The network architecture consists of an initial convolutional layer followed by a series of convolutional blocks. Each block contains a convolutional layer, an instance

normalization layer, and a LeakyReLU activation function. The number of output channels for each block is gradually increased from 64 to 512, and the stride is set to 2 for all blocks except the last one. The final output of the network is obtained by passing the output of the last block through a convolutional layer with a kernel size of 4 and stride 1, followed by a sigmoid activation function to squash the output to the range [0, 1].

The test() function creates a random tensor of shape (5, 3, 256, 256) and passes it through the discriminator network, printing the shape of the output tensor.

6. Loss functions:

a) Loss functions TensorFlow:

Real X = Horse, Fake Y = generated Zebra, Cycled X = Horse generated from generated Zebra, Real Y = Zebra, Fake X = generated Horse, Cycled Y = Zebra generated from generated Horse, Same X = generated Horse from Real X, Same Y = generated Zebra from Real Y, Disc real Y = discriminator output with real Zebra as input, Disc real X = discriminator output with real Horse as input, Disc Fake X = discriminator output with generated Horse as input , Disc Fake Y = discriminator output with generated Zebra as input, Generator G loss = generator loss for DISC fake Y, Generator Floss = generator loss for DISC fake X,
Identity loss is between generated Zebra with Zebras as input to generator and zebra and Identity loss is between generated Horse with Horse as input to generator and Horse.

Hence,

Total cycle loss = Cycle loss (between real horse and generated horse) + Cycle loss (between real Zebra and generated Zebra)

Total Generator G loss = Generator G loss + Total cycle loss + Identity loss(Real Zebra. Generated Zebra)
Total Generator F loss = Generator F loss + Total cycle loss + Identity loss(Real Horse. Generated Horse)

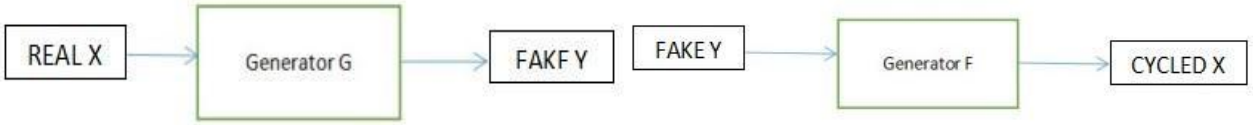
Discriminator X loss = loss between Disc real X and Disc fake X
Discriminator Y loss = loss between Disc real Y and Disc fake Y

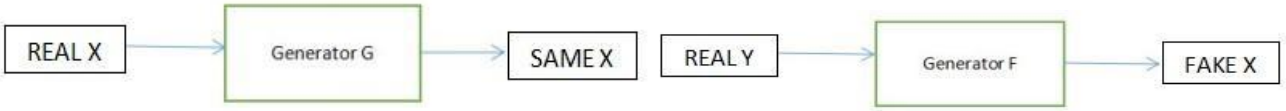
Cycle loss - Calculates cycle consistency loss between real image and cycled image. It calculates absolute difference between real and cycled image and then takes mean of absolute difference. Then it is multiplied by hyper parameter lambda which is basically a weighted parameter.

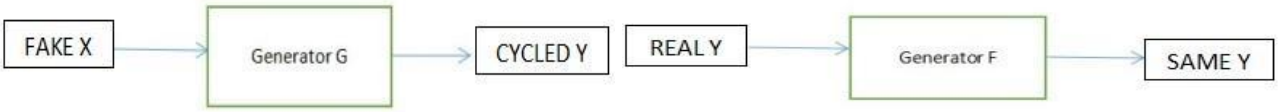
Generator_loss (Generator G loss ,Generator F loss) - The function generator_loss calculates the binary cross-entropy loss between a target of all ones and the generated output, where the binary_loss function is used with from_logits=True indicating that the generated output is not normalized using a sigmoid activation function. The calculated loss represents the discrepancy between the generated output and the target of all ones, and it is used in the optimization process of the generator model to improve its performance during training.

Discriminator loss (Discriminator X loss, Discriminator Y loss) - The function discriminator_loss calculates the total loss for a discriminator model in a binary classification task, where it distinguishes between real and generated data. The binary_loss function is used with from_logits=True indicating that the discriminator outputs are not normalized using a sigmoid activation function. The real loss is computed by comparing the real data (real) to a target of all ones using the binary_loss function, while the generated loss is computed by comparing the generated data (generated) to a target of all zeros. The two losses are summed up to get the total discriminator loss, which is then multiplied by 0.5 to enforce a balanced contribution from the real and generated losses. The calculated loss represents the discrepancy between the discriminator's predictions and the target labels, and it is used in the overall optimization process of the discriminator model during training to improve its ability to distinguish between real and generated data.

Identity loss - The function identity_loss calculates the identity loss between a real image and its corresponding same-image, where the same-image is generated by the same image-to-image translation model. It first computes the absolute difference between the real image and the same-image, and then takes the mean of this absolute difference. The resulting value is multiplied by LAMBDA * 0.5 to determine the final identity loss, which represents the discrepancy between the real image and the same-image. This loss is used in the optimization process of the image-to-image translation model during training, with the aim of preserving the identity of the input images in the generated outputs.


REAL X
Generator G
FAKF Y
FAKE Y
Generator F
CYCLED X


REAL X
Generator G
SAME X
REAL Y
Generator F
FAKE X

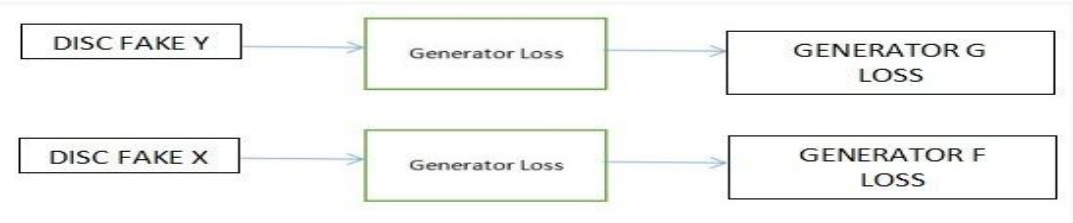

FAKE X
Generator G
CYCLED Y
REAL Y
Generator F
SAME Y


REAL X
Descriminator X
DISC REAL X
REAL Y
Descriminator Y
DISC REAL Y


FAKE X
Descriminator Y
DISC FAKE X
FAKE Y
Descriminator Y
DISC FAKE Y
Desc Real X
Desc fake X
Descriminator G Loss
Desc X Loss
Desc real Y
Desc fake Y
Descriminator F Loss
Desc Y Loss

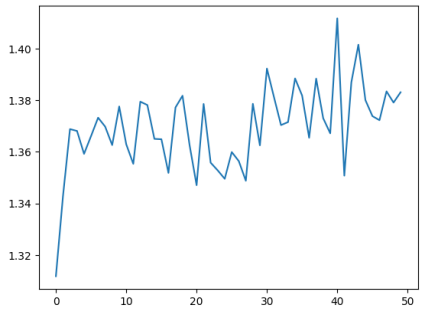

REAL X
CYCLED X
Cycle loss
REAL Y
CYCLED Y
Cycle loss
TOTAL CYCLE LOSS
REAL X
Descriminator X
DISC REAL X


REAL X
SAME X
Identity loss
REAL Y
SAME Y
Identity loss

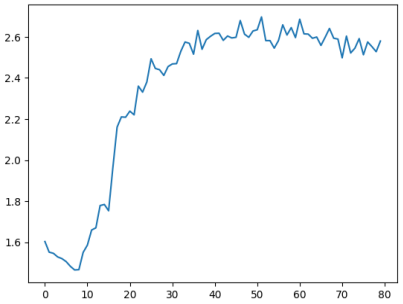

DISC FAKE Y
Generator Loss
GENERATOR G LOSS
DISC FAKE X
Generator Loss
GENERATOR F LOSS

Total loss graphs of 4 models generated during training:

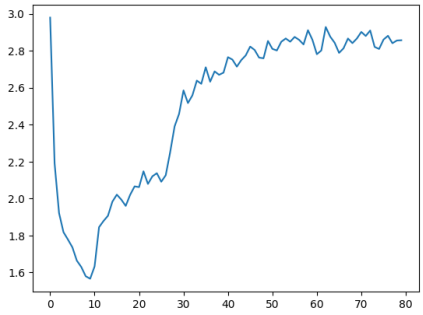
Model 1:



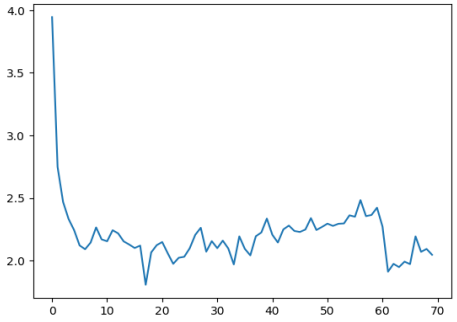
Model 2:



Model 3:



Model 4:



b) Loss functions (Pytorch) :

The loss function in this model is a combination of several loss functions that are used to train the Generators and Discriminators in the CycleGAN architecture.

The Discriminator loss function is defined using Mean Squared Error (MSE) loss, which compares the output of the Discriminator with the ground truth labels (real/fake). The losses of both Discriminators (D_H and D_Z) are combined by taking the average of both losses.

The Generator loss function is defined using three types of losses. The first loss is an adversarial loss, which encourages the Generator to produce images that can fool the Discriminator into thinking they are real. The second loss is a cycle consistency loss, which ensures that the reconstructed image after passing through the Generator and the inverse Generator is close to the original image. Finally, the third loss is an identity loss, which ensures that an image from one domain (e.g., horse) remains unchanged when passed through its corresponding Generator.

The losses are combined to form the final loss for the Generators by summing up the adversarial losses, cycle consistency losses, and identity losses, weighted by their respective hyperparameters. The Generator loss is optimized using Adam optimizer. Additionally, mixed precision training is used to speed up the training process.

The adversarial loss is used to fool the discriminator by tricking it into believing that the fake image generated by the generator is a real image. The generator's goal is to fool the discriminator, so the loss function for the generator is mse(dz_fake, 1), where dz_fake is the output of the discriminator when given the generated fake image.

The cycle loss ensures that the generated image is the same as the original image. The cycle loss is calculated by passing the fake image through the generator in the opposite direction and comparing it to the original image.

The identity loss is used to ensure that the generator does not change the image too much. This is done by comparing the original image to the output of the generator when it is given the original image as input.

The final loss is a combination of all three losses, where each loss is multiplied by a corresponding weighting factor, lambda_cycle for cycle loss, and lambda_identity for identity loss. Finally, the generator is updated using the Adam optimizer with the backward function.

It should be noted that the identity loss was not used in the paper that this implementation is based on.

7. Data pre processing:

a) Image augmentation:

Random flip left and right - Flip the image randomly left or right on horizontal axis
Random brightness - randomly increase or decrease brightness of image
Random contrast - randomly increase or decrease contrast of image
Random hue - apply a random shift or rotation to the hue values of the pixels in the image
Random saturation - apply randomness to the saturation component of an image
Random crop - randomly crop image of dimension 256*256*3 given an image of input dimension 286*286*3

b) Image normalization:

Image is normalized by casting image data type to float32, then dividing image by 127.5 and subtracting by 1. By doing this image pixel values will be in range -1 to 1.

8. Training parameters:

Batch size: 1-4
Epochs: 50 - 80
Optimizer: Adam
Input image dimension - 256*256*3
Learning rate - 0.0002
Normalization : Instance normalization

9. Model evaluation (Tensor Flow):

Evaluation metric - Generally FID is used as evaluation metric in GAN. FID (Fréchet Inception Distance): is a measure of how similar are generated and ground-truth image. But in our cases we do not have specific ground truth images, here there is no direct correspondence between images of source and target domain. Here we use cycle consistency loss as a metric to ensure translated images are consistent when translated back to original domain. SSIM and MSE are evaluation metrics used in GANs to measure the performance of the generator network. SSIM measures the structural similarity between the generated and real images, while absolute mean square error measures the average difference between the pixel values of the two images.

Four models are trained:

Model 1: Generator - custom_unet_generator_v2, Discriminator - pix2pix discriminator
Model 2: Generator - custom_unet_generator_v2, Discriminator - custom_unet_descriminator_dilated_v2

Model 3: Generator - custom_unet_generator_v2, Discriminator - custom_unet_descriminator_v2
Model 4: Generator - custom_unet_generator_v3 ,Discriminator - custom_unet_descriminator_v2

Model	Cycle loss	Absolute mean square error (0 to +ve infinity)	SSIM (-1 to 1)
Model 1	Horse to Zebra and Zebra to Horse	0.5240	0.91758
	Zebra to Horse and Horse to Zebra	0.8340	0.92381

Model 2	Horse to Zebra and Zebra to Horse Zebra to Horse and Horse to Zebra	1.1713 1.8640	0.7610 0.6935
Model 3	Horse to Zebra and Zebra to Horse Zebra to Horse and Horse to Zebra	0.6822 0.9173	0.8768 0.4908
Model 4	Horse to Zebra and Zebra to Horse Zebra to Horse and Horse to Zebra	1.1035 0.9173	0.7625 0.7078

Model evaluation (Pytorch):

Evaluation Metric	Value on Horse Images	Value on Generated Images
H_real	0.75	0.19
H_fake	0.17	0.78

H_real represents the mean output of the discriminator H on real horse images, calculated during training. This metric measures how well the discriminator is able to distinguish between real and fake horse images, with a higher value indicating that the discriminator is better at identifying real images.

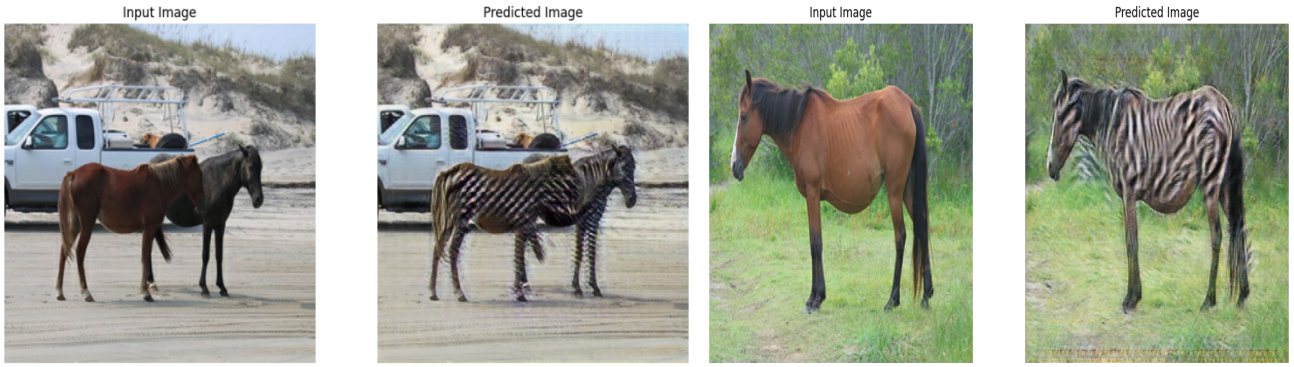
H_fake represents the mean output of the discriminator H on fake horse images generated by the generator H, calculated during training. This metric measures how well the generator is able to produce images that can fool the discriminator. In this case, the value of 0.17 indicates that the discriminator is able to distinguish between fake horse images generated by the generator and real horse images, meaning that the generator is not yet producing images that are convincingly realistic. The generator will continue to improve as its output starts to more closely resemble real horse images, leading to a higher value for H_fake.

The goal of training a CycleGAN model is to minimize the discrepancy between the distributions of images in the two domains being translated, so the aim is to achieve values of H_real close to 1 and H_fake close to 0.5. In practice, it is unlikely to achieve these ideal values, but the values obtained during training can still be used to assess the quality of the model's outputs.

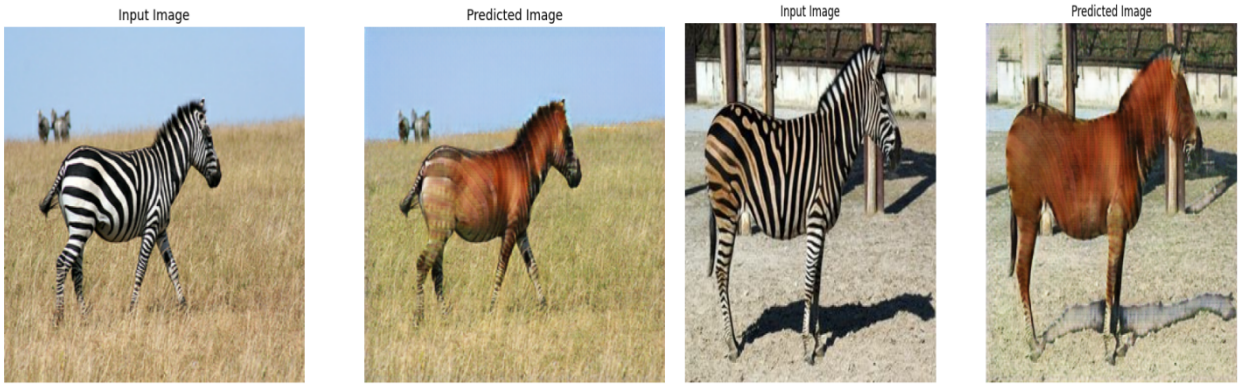
These metrics are logged using the tqdm package and printed at the end of each training epoch. The evaluation of the model is done by computing various loss functions such as L1 and MSE loss. The model is trained using the Adam optimizer with a learning rate of 0.0002 and beta1=0.5, beta2=0.999. The model checkpoints are saved after every epoch using save_checkpoint() function in the utils.py file. To load a saved checkpoint, the load_checkpoint() function is used. The progress of the training is shown using the tqdm library.

10. Results (Tensor Flow):

Horse to Zebra:



Zebra to Horse:



Results (Pytorch):

Zebra to Horse:

Horse to Zebra:

<u>Note</u>: Since we had limited computational resources (GPU), we were unable to train models with bigger architectures. We trained all our models on Google Colab, but the platform was unreliable as it frequently disconnected during training. Despite this problem, we continued to experiment with different architectures with lower image resolution.

11. Contribution

Task	Naveen Raju Sreerama Raju Govinda Raju				Jai Raj Choudhary
Problem statement analysis and Research	50				50
Coding (Generators, Discriminators, Augmentation pipeline, Model Evaluation code)	Model1	Model 2	Model 3	Model 4	Model(Pytorch) (training, testing)
	100	100	100	100	100
Debugging and Testing	Model1	Model 2	Model 3	Model 4	Model
	100	100	100	100	100
PPT	70				30
Report	70				30

12. Bibliography

- https://www.researchgate.net/publication/323904616_Patch-Based_Image_Inpainting_with_Generative_Adversarial_Networks (Patch-Based Image Inpainting with Generative Adversarial Networks)
- https://arxiv.org/pdf/1505.04597.pdf (U-Net: Convolutional Networks for Biomedical Image Segmentation)
- https://arxiv.org/pdf/1703.10593.pdf (Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks)
- https://www.tensorflow.org/tutorials/generative/pix2pix (pix2pix GAN)
- https://github.com/tensorflow/examples/blob/master/tensorflow_examples/models/pix2pix/pix2pix.py (Instance normalisation)
- https://www.tensorflow.org/tutorials/generative/cyclegan (Cycle GAN)
- https://hardikbansal.github.io/CycleGANBlog/ (Cycle GAN Blog)
- Official PyTorch implementation of CycleGAN: https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix
- PyTorch implementation of CycleGAN with additional features: https://github.com/aitorzip/PyTorch-CycleGAN
- PyTorch implementation of CycleGAN with progressive training: https://github.com/taesungp/pytorch-cyclegan-pix2pix
- PyTorch implementation of CycleGAN with spectral normalization: https://github.com/christiancosgrove/pytorch-spectral-normalization-gan