Maynooth University
National University of Ireland Maynooth

Machine Learning and Neural Networks
(CS401)

Project Topic

CYCLE-GAN
(Generative Adversarial Network)

Project link

Project Team

Student Name	Student Number
Jigar Patel	21251312
Lavy Tejraj Asnani	21250224
Naresh Kumar Nagaraj	21250702

INTRODUCTION

Generative Adversarial Networks (GAN) are used in a variety of fields like to convert X ray images into MRI scan and vice versa, satellite images into google map like images and vice versa, season transfer images, object transformation of images like from apples to oranges and vice versa or zebra-horses and vice versa, etc. Here, in our project we are implementing an image-to-image translation using CycleGAN (Cycle-Consistent Generative Adversarial Networks) where we will only change the pattern on the skin of the horse and keep other factors like the background, shape of the horse, etc. totally unchanged. CycleGAN uses a cycle consistency loss to enable training without the need for paired data. In other words, it can translate from one domain to another without a one-to-one mapping between the source and target domain. In this project, we have created and trained a CycleGAN model to transform horses into zebras. We'll start by looking at the structure of a CycleGAN and the four loss functions that it uses to train. We'll get a look at the architecture of each discriminator and generator in a CycleGAN as well. The essential concept of a GAN is "indirect" training via a discriminator, which is another neural network that can determine how realistic an input is and is updated constantly as well. This essentially means that the generator is trained to deceive the discriminator rather than minimize the distance to a certain image. This allows the model to learn without being supervised. Finally, we'll go over how to train a CycleGAN, with some really remarkable translation results at the conclusion.

Tools and Libraries

The application is made up of several libraries that employ a wide range of tools and datasets. They're all crucial to the creation of the overall application, including the optional interfaces. Some of the libraries, tools, and datasets that can be used with it are listed below.

- TensorFlow: it is an open-source framework for creating models with large-scale, multi-layer networks.
- Keras: it is an open-source neural-network library that allows users to create products using deep models. It is used in the application to predict how patients would feel.
- PIL: it is a Python bindings library focused at import image.
- NumPy: it is a Python package that includes multidimensional array objects as well as array processing functions.
- Matplotlib: it is a comprehensive library for creating static, animated, and interactive visualizations in Python.
- Python: it is a general-purpose programming language with a high-level interpreter. It's garbage-collected, and it's typed dynamically. It can handle a wide range of programming paradigms, including structured object-oriented programming and functional programming. It includes a sizable standard library. The application is written in this programming language.
- IPython: it is a command shell for interactive computing in multiple programming languages

Datasets

An unpaired-images dataset for training CycleGANs - horse-zebra, map-image, etc. (https://www.kaggle.com/suyashdamle/cyclegan). The length of the training set is around 1067 and the length of the test set is around 100. Each image is of size- 128 x 128. It took 3 days to train the data and the results would have been better if trained for a longer duration.

DEVELOPED METHODOLOGY

Cycle-GAN

The model architecture is made up of two generator models: one (Generator-F) for generating images for the first domain (Domain-X), and another (Generator-G) for generating images for the second domain (Domain-Y).

- Generator-F -> Domain-X (horse images)
- Generator-G -> Domain-Y (zebra images)

The picture production process is conditional on an input image, specifically an image from the other domain, as the generator models execute image translation.

- Domain-Y(zebra) -> Generator-F -> Domain-X(horse)
- Domain-X(horse) -> Generator-G -> Domain-Y(zebra)

A discriminator model corresponds to each generator. The first discriminator model (Discriminator-X) compares real and created images from Domain-X and determines whether they are authentic or false. The second discriminator model (Discriminator-Y) compares real and created images from Domain-Y and determines whether they are authentic or false.

- Domain-X -> Discriminator-X -> [Real/Fake]
- Domain-Y -> Generator-F -> Discriminator-X -> [Real/Fake]
- Domain-Y -> Discriminator-Y -> [Real/Fake]
- Domain-X -> Generator-G -> Discriminator-Y -> [Real/Fake]

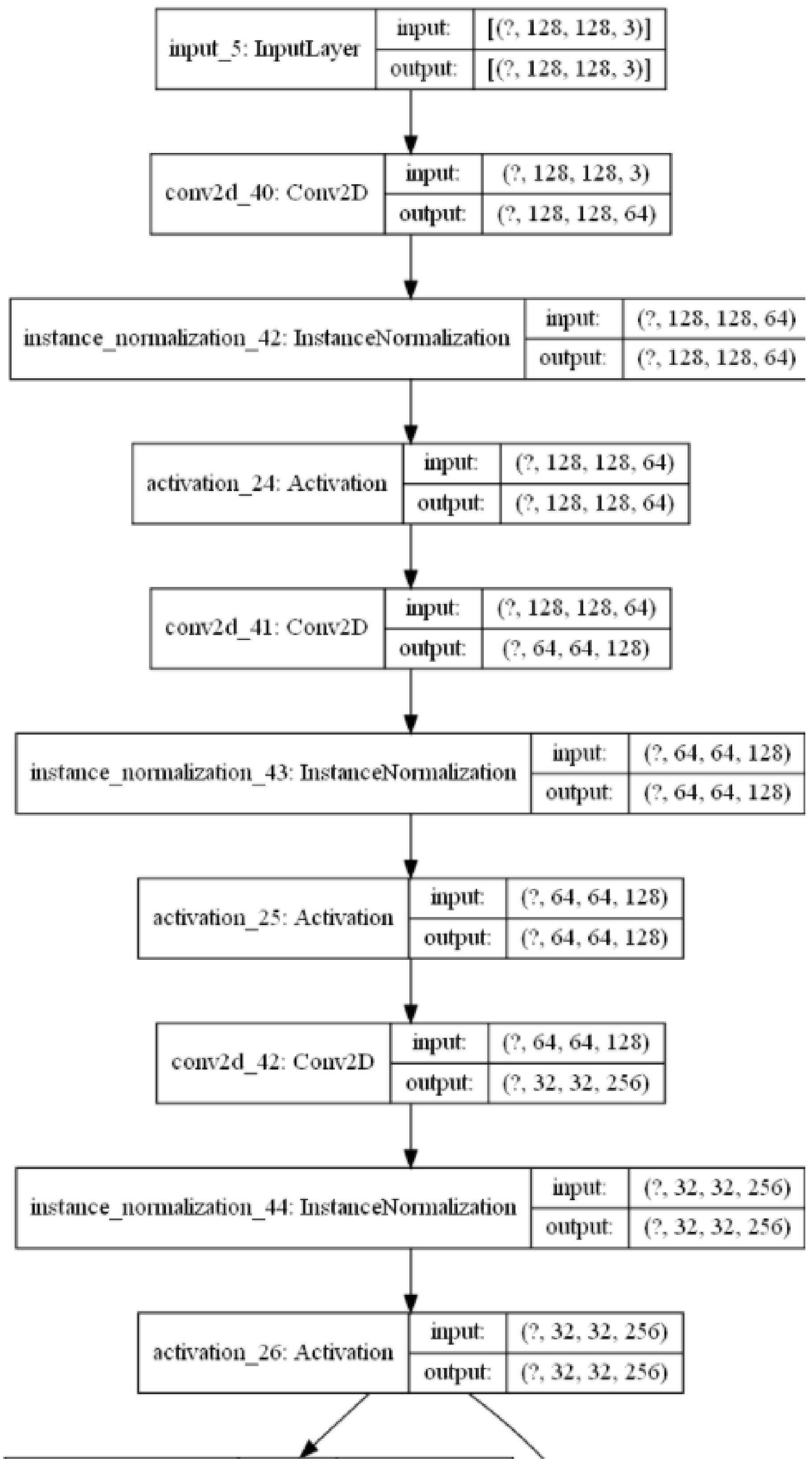
Like standard GAN models, the discriminator and generator models are trained in an adversarial zero-sum process. The generators improve their ability to deceive the discriminators, and the discriminators improve their ability to recognize bogus images. During the training phase, the models work together to find a balance. Furthermore, the generator models are regularized to create translated versions of the source domain's input images rather than fresh images in the destination domain. This is accomplished by feeding generated photos into the appropriate generator model and comparing the resultant image to the original. A cycle is when an image is sent through both generators. Cycle consistency is achieved by training each pair of generator models to better mimic the original source image.

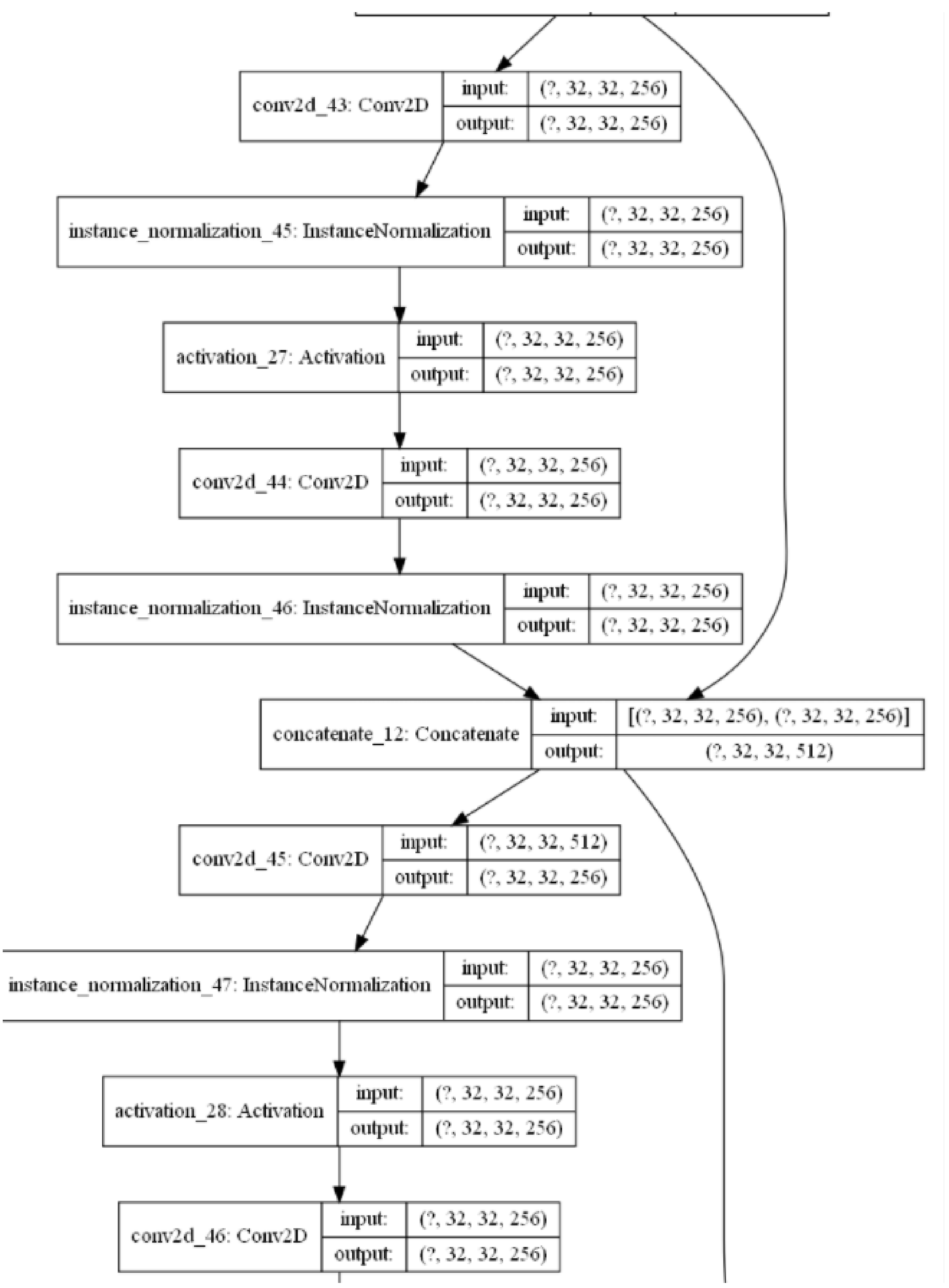
- Domain-Y -> Generator-F -> Domain-X -> Generator-G -> Domain-Y
- Domain-X -> Generator-G -> Domain-Y -> Generator-F -> Domain-X

Now that we've learned about the model architecture, we can look at each model individually to see how it may be implemented.

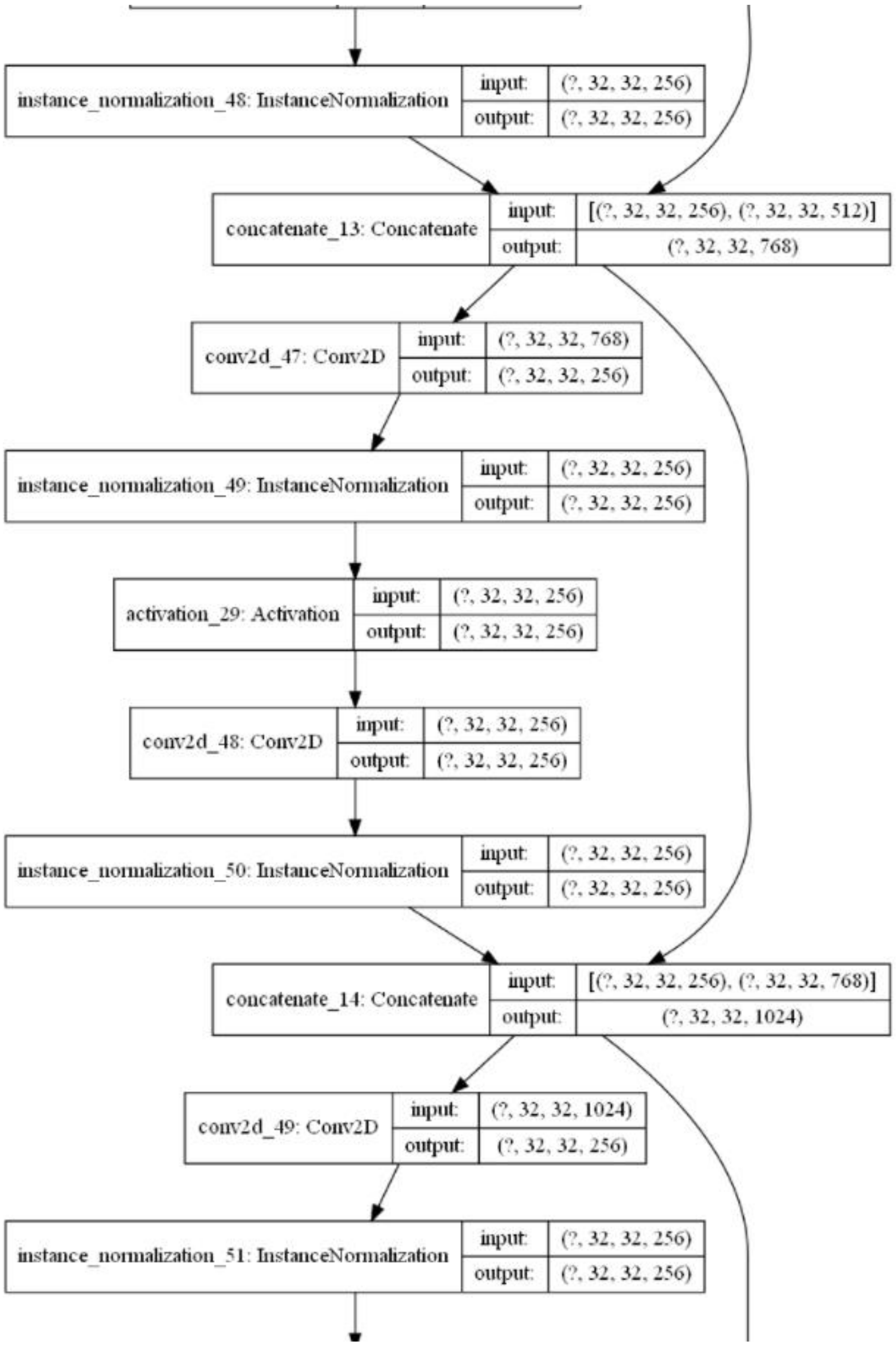
CycleGAN Generator Model

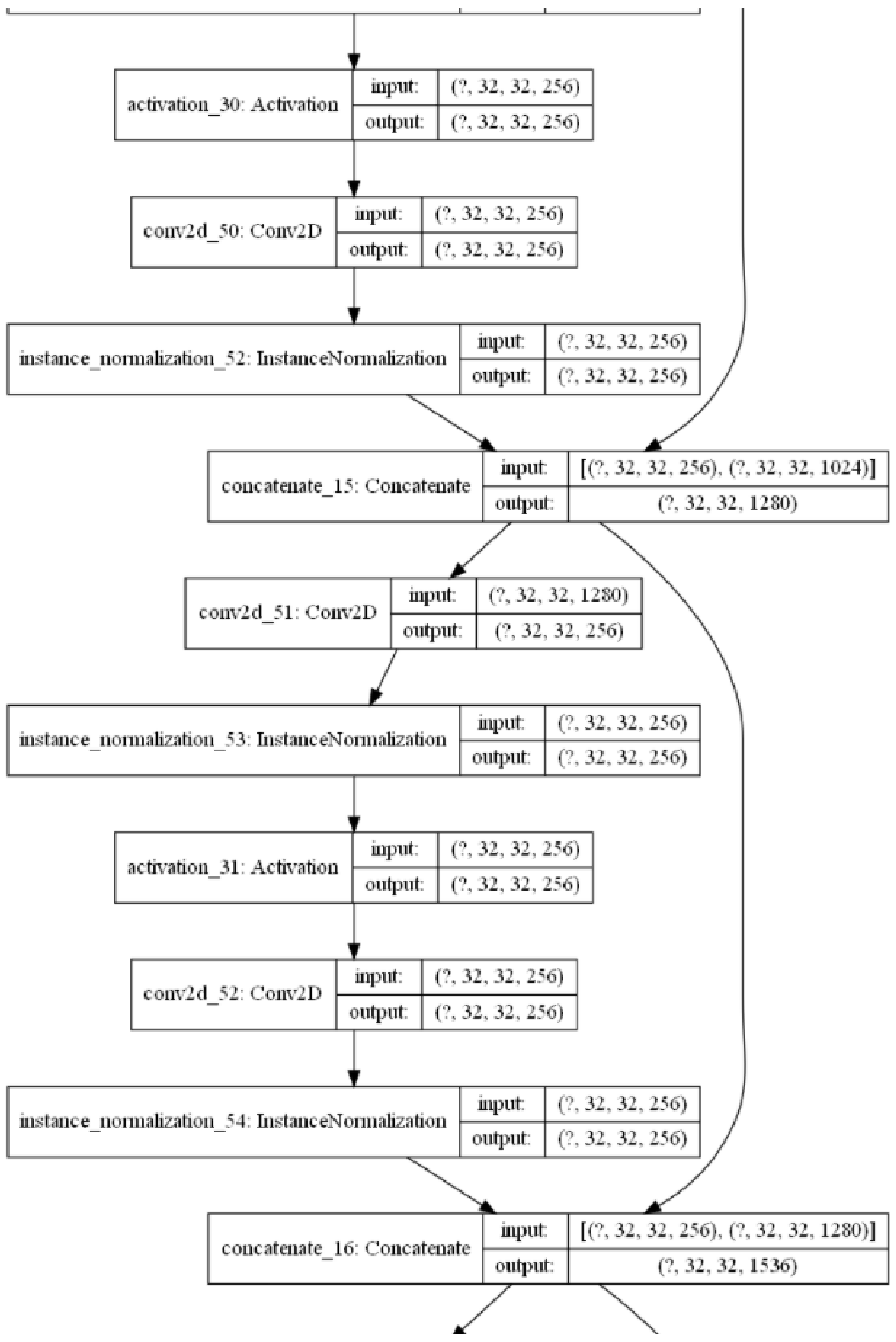
The CycleGAN Generator model receives an image as input and outputs a translated image. The model encodes the input image with a series of down sampling convolutional blocks, transforms the image with a series of residual network (ResNet) convolutional blocks, and generates the output image with a series of up sampling convolutional blocks.

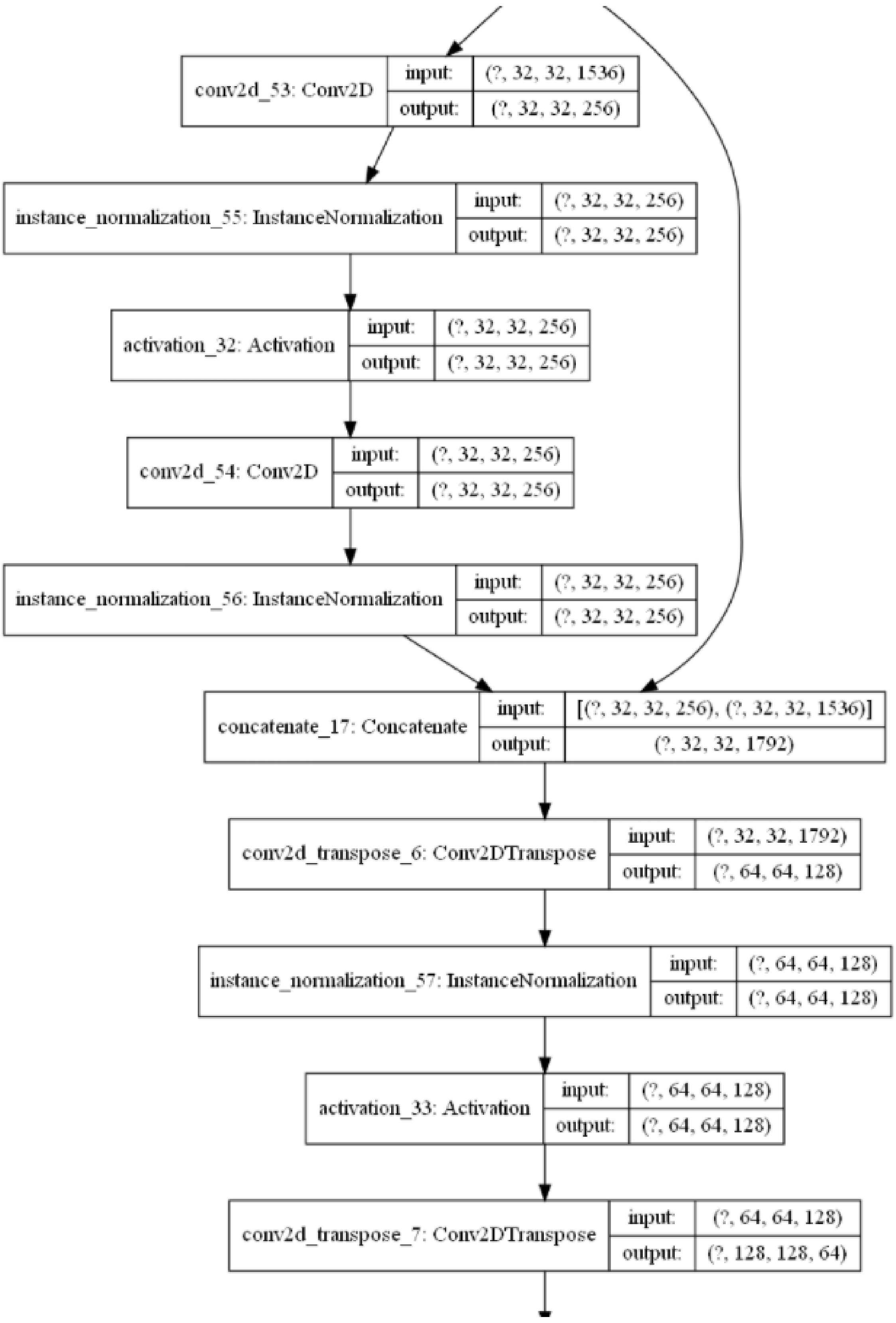

input_5: InputLayer
input:
[(?, 128, 128, 3)]
output:
[(?, 128, 128, 3)]
conv2d_40: Conv2D
input:
(?, 128, 128, 3)
output:
(?, 128, 128, 64)
instance_normalization_42: InstanceNormalization
input:
(?, 128, 128, 64)
output:
(?, 128, 128, 64)
activation_24: Activation
input:
(?, 128, 128, 64)
output:
(?, 128, 128, 64)
conv2d_41: Conv2D
input:
(?, 128, 128, 64)
output:
(?, 64, 64, 128)
instance_normalization_43: InstanceNormalization
input:
(?, 64, 64, 128)
output:
(?, 64, 64, 128)
activation_25: Activation
input:
(?, 64, 64, 128)
output:
(?, 64, 64, 128)
conv2d_42: Conv2D
input:
(?, 64, 64, 128)
output:
(?, 32, 32, 256)
instance_normalization_44: InstanceNormalization
input:
(?, 32, 32, 256)
output:
(?, 32, 32, 256)
activation_26: Activation
input:
(?, 32, 32, 256)
output:
(?, 32, 32, 256)

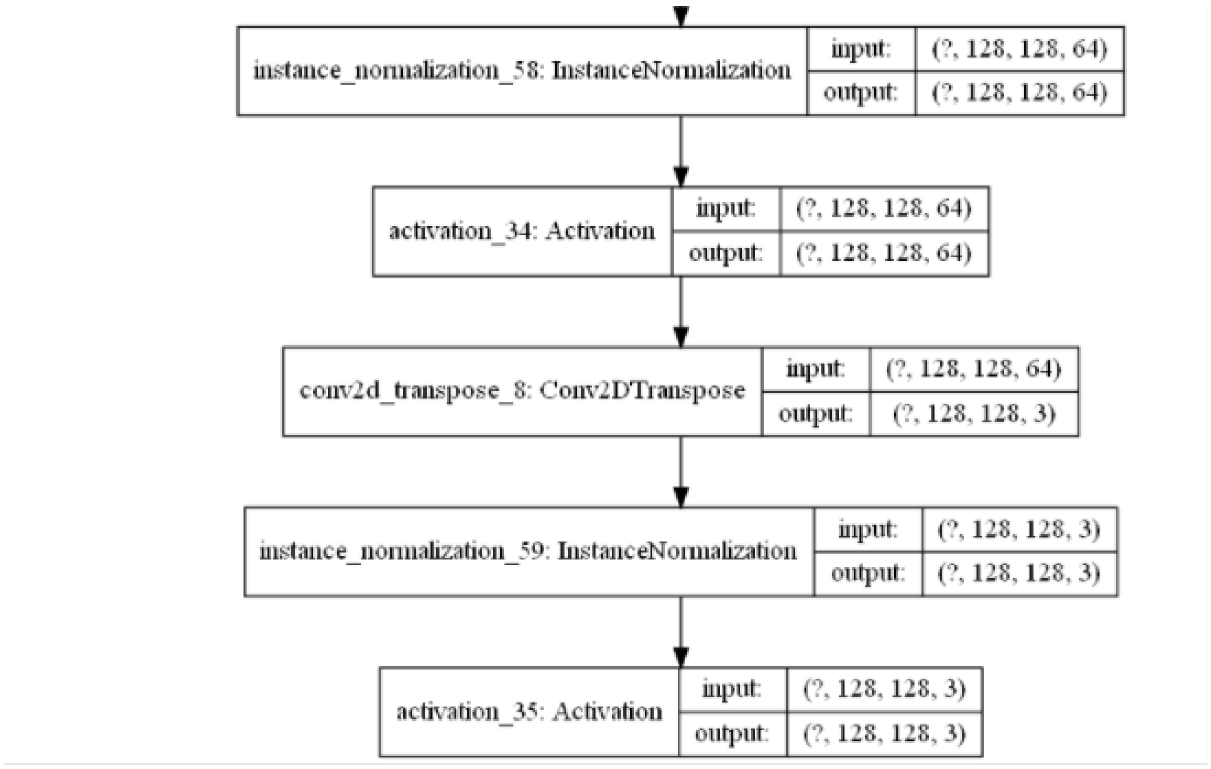


conv2d_43: Conv2D | input: (?, 32, 32, 256) | output: (?, 32, 32, 256)
instance_normalization_45: InstanceNormalization | input: (?, 32, 32, 256) | output: (?, 32, 32, 256)
activation_27: Activation | input: (?, 32, 32, 256) | output: (?, 32, 32, 256)
conv2d_44: Conv2D | input: (?, 32, 32, 256) | output: (?, 32, 32, 256)
instance_normalization_46: InstanceNormalization | input: (?, 32, 32, 256) | output: (?, 32, 32, 256)
concatenate_12: Concatenate | input: [(?, 32, 32, 256), (?, 32, 32, 256)] | output: (?, 32, 32, 512)
conv2d_45: Conv2D | input: (?, 32, 32, 512) | output: (?, 32, 32, 256)
instance_normalization_47: InstanceNormalization | input: (?, 32, 32, 256) | output: (?, 32, 32, 256)
activation_28: Activation | input: (?, 32, 32, 256) | output: (?, 32, 32, 256)
conv2d_46: Conv2D | input: (?, 32, 32, 256) | output: (?, 32, 32, 256)


instance_normalization_48: InstanceNormalization
input: (?, 32, 32, 256)
output: (?, 32, 32, 256)
concatenate_13: Concatenate
input: [(?, 32, 32, 256), (?, 32, 32, 512)]
output: (?, 32, 32, 768)
conv2d_47: Conv2D
input: (?, 32, 32, 768)
output: (?, 32, 32, 256)
instance_normalization_49: InstanceNormalization
input: (?, 32, 32, 256)
output: (?, 32, 32, 256)
activation_29: Activation
input: (?, 32, 32, 256)
output: (?, 32, 32, 256)
conv2d_48: Conv2D
input: (?, 32, 32, 256)
output: (?, 32, 32, 256)
instance_normalization_50: InstanceNormalization
input: (?, 32, 32, 256)
output: (?, 32, 32, 256)
concatenate_14: Concatenate
input: [(?, 32, 32, 256), (?, 32, 32, 768)]
output: (?, 32, 32, 1024)
conv2d_49: Conv2D
input: (?, 32, 32, 1024)
output: (?, 32, 32, 256)
instance_normalization_51: InstanceNormalization
input: (?, 32, 32, 256)
output: (?, 32, 32, 256)

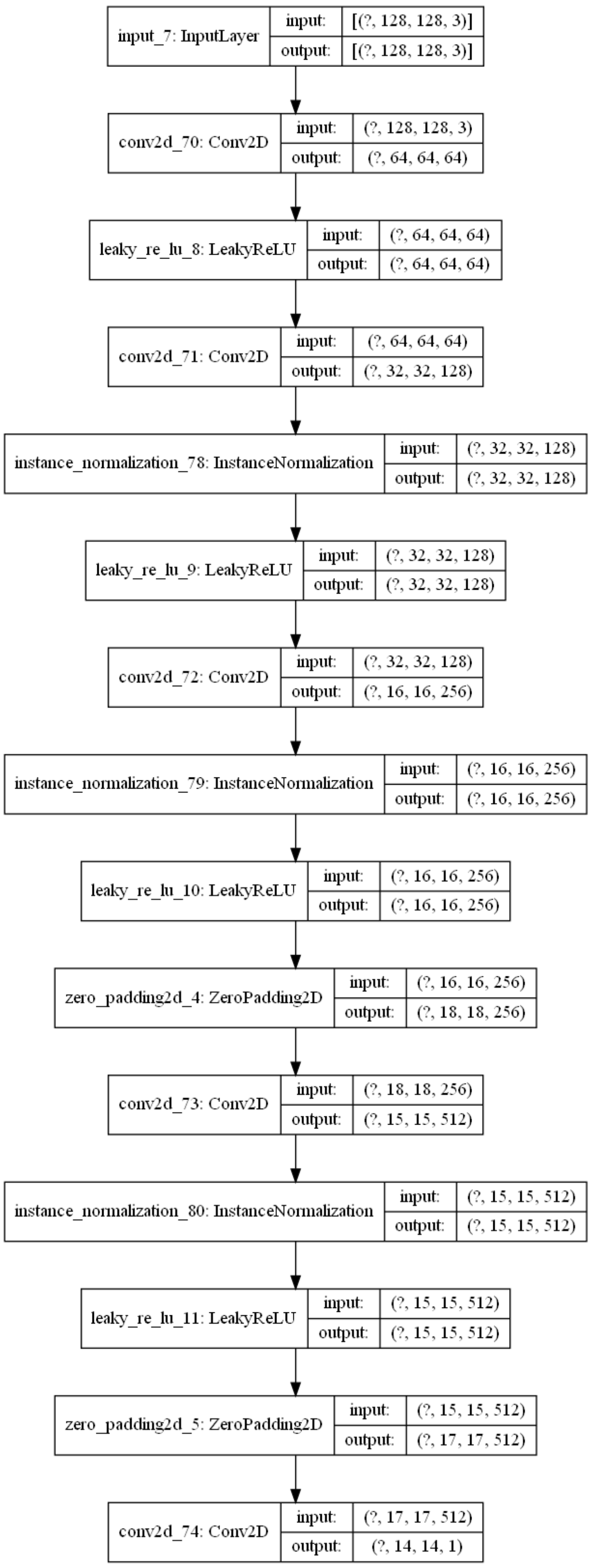

activation_30: Activation
input: (?, 32, 32, 256)
output: (?, 32, 32, 256)
conv2d_50: Conv2D
input: (?, 32, 32, 256)
output: (?, 32, 32, 256)
instance_normalization_52: InstanceNormalization
input: (?, 32, 32, 256)
output: (?, 32, 32, 256)
concatenate_15: Concatenate
input: [(?, 32, 32, 256), (?, 32, 32, 1024)]
output: (?, 32, 32, 1280)
conv2d_51: Conv2D
input: (?, 32, 32, 1280)
output: (?, 32, 32, 256)
instance_normalization_53: InstanceNormalization
input: (?, 32, 32, 256)
output: (?, 32, 32, 256)
activation_31: Activation
input: (?, 32, 32, 256)
output: (?, 32, 32, 256)
conv2d_52: Conv2D
input: (?, 32, 32, 256)
output: (?, 32, 32, 256)
instance_normalization_54: InstanceNormalization
input: (?, 32, 32, 256)
output: (?, 32, 32, 256)
concatenate_16: Concatenate
input: [(?, 32, 32, 256), (?, 32, 32, 1280)]
output: (?, 32, 32, 1536)


conv2d_53: Conv2D
input: (?, 32, 32, 1536)
output: (?, 32, 32, 256)
instance_normalization_55: InstanceNormalization
input: (?, 32, 32, 256)
output: (?, 32, 32, 256)
activation_32: Activation
input: (?, 32, 32, 256)
output: (?, 32, 32, 256)
conv2d_54: Conv2D
input: (?, 32, 32, 256)
output: (?, 32, 32, 256)
instance_normalization_56: InstanceNormalization
input: (?, 32, 32, 256)
output: (?, 32, 32, 256)
concatenate_17: Concatenate
input: [(?, 32, 32, 256), (?, 32, 32, 1536)]
output: (?, 32, 32, 1792)
conv2d_transpose_6: Conv2DTranspose
input: (?, 32, 32, 1792)
output: (?, 64, 64, 128)
instance_normalization_57: InstanceNormalization
input: (?, 64, 64, 128)
output: (?, 64, 64, 128)
activation_33: Activation
input: (?, 64, 64, 128)
output: (?, 64, 64, 128)
conv2d_transpose_7: Conv2DTranspose
input: (?, 64, 64, 128)
output: (?, 128, 128, 64)


instance_normalization_58: InstanceNormalization
input: (?, 128, 128, 64)
output: (?, 128, 128, 64)
activation_34: Activation
input: (?, 128, 128, 64)
output: (?, 128, 128, 64)
conv2d_transpose_8: Conv2DTranspose
input: (?, 128, 128, 64)
output: (?, 128, 128, 3)
instance_normalization_59: InstanceNormalization
input: (?, 128, 128, 3)
output: (?, 128, 128, 3)
activation_35: Activation
input: (?, 128, 128, 3)
output: (?, 128, 128, 3)

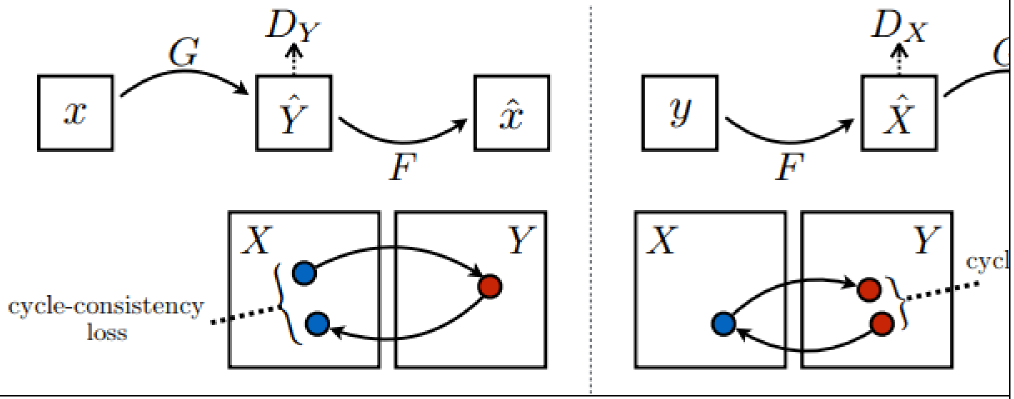
CycleGAN Discriminator Model

The discriminator model is in charge of predicting whether a picture is real or fake using a real or created image as input.


input_7: InputLayer | input: [(?, 128, 128, 3)] | output: [(?, 128, 128, 3)]
conv2d_70: Conv2D | input: (?, 128, 128, 3) | output: (?, 64, 64, 64)
leaky_re_lu_8: LeakyReLU | input: (?, 64, 64, 64) | output: (?, 64, 64, 64)
conv2d_71: Conv2D | input: (?, 64, 64, 64) | output: (?, 32, 32, 128)
instance_normalization_78: InstanceNormalization | input: (?, 32, 32, 128) | output: (?, 32, 32, 128)
leaky_re_lu_9: LeakyReLU | input: (?, 32, 32, 128) | output: (?, 32, 32, 128)
conv2d_72: Conv2D | input: (?, 32, 32, 128) | output: (?, 16, 16, 256)
instance_normalization_79: InstanceNormalization | input: (?, 16, 16, 256) | output: (?, 16, 16, 256)
leaky_re_lu_10: LeakyReLU | input: (?, 16, 16, 256) | output: (?, 16, 16, 256)
zero_padding2d_4: ZeroPadding2D | input: (?, 16, 16, 256) | output: (?, 18, 18, 256)
conv2d_73: Conv2D | input: (?, 18, 18, 256) | output: (?, 15, 15, 512)
instance_normalization_80: InstanceNormalization | input: (?, 15, 15, 512) | output: (?, 15, 15, 512)
leaky_re_lu_11: LeakyReLU | input: (?, 15, 15, 512) | output: (?, 15, 15, 512)
zero_padding2d_5: ZeroPadding2D | input: (?, 15, 15, 512) | output: (?, 17, 17, 512)
conv2d_74: Conv2D | input: (?, 17, 17, 512) | output: (?, 14, 14, 1)

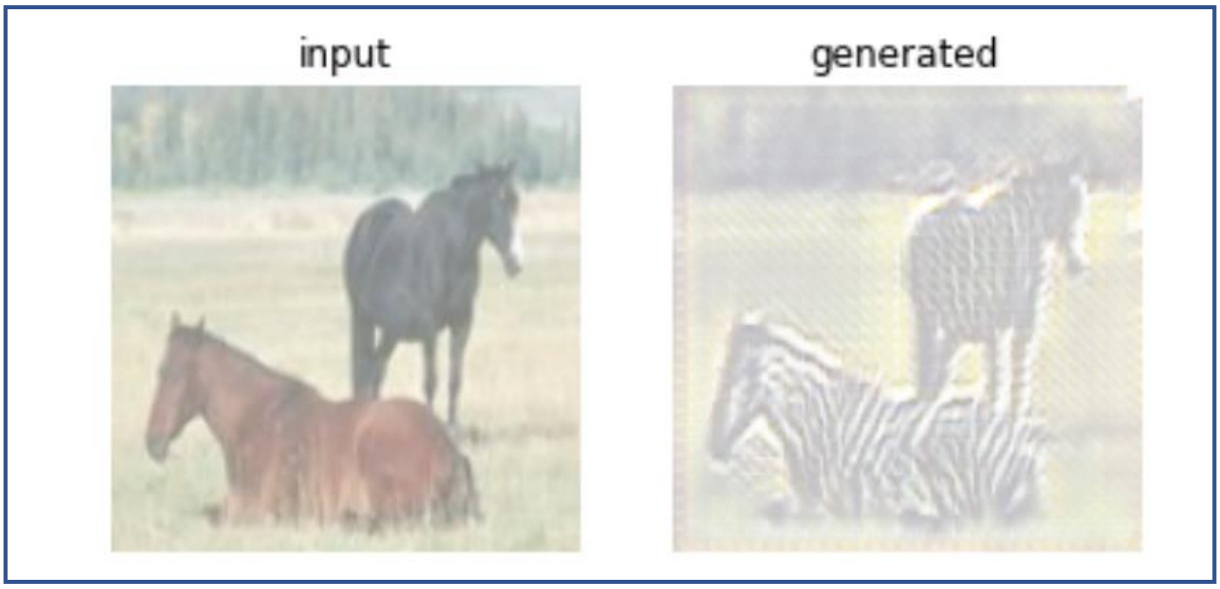
Losses:

There are four loss function being simultaneously used.

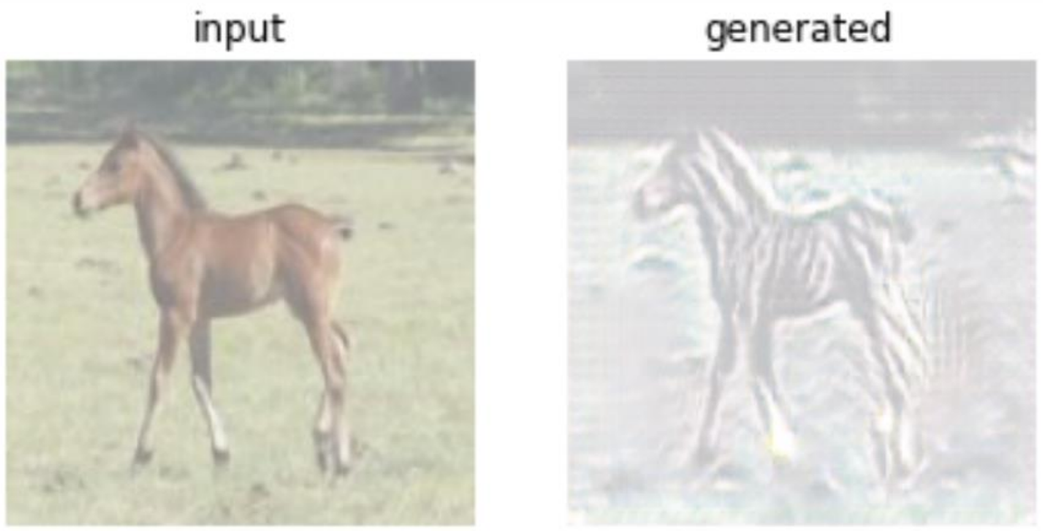
Discriminator loss	This makes sure the discriminators correctly identify generated images as fake images.
Generator loss	This makes sure the generators produce images which can trick the discriminator into considering them as real images.
Cycle consistency loss	Compares X with generated X.  D_Y G x $\hat{Y}$ $\hat{x}$ F X Y cycle-consistency loss D_X y $\hat{X}$ F X Y cycl
Identity loss	|G(Y)-Y| + |F(X)-X|

EXPERIMENTAL RESULTS

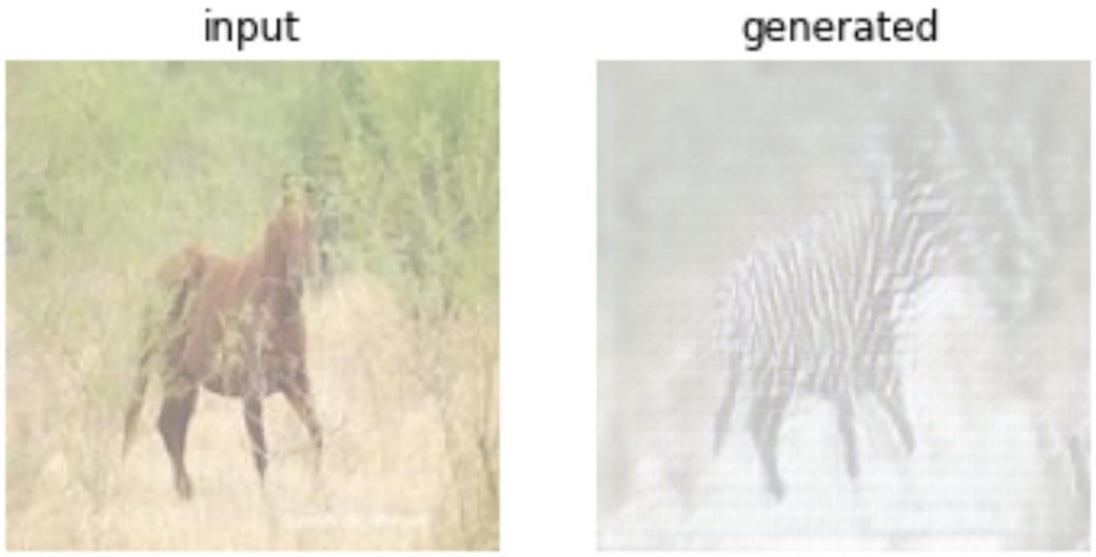
Below screenshot shows the output after running the model:

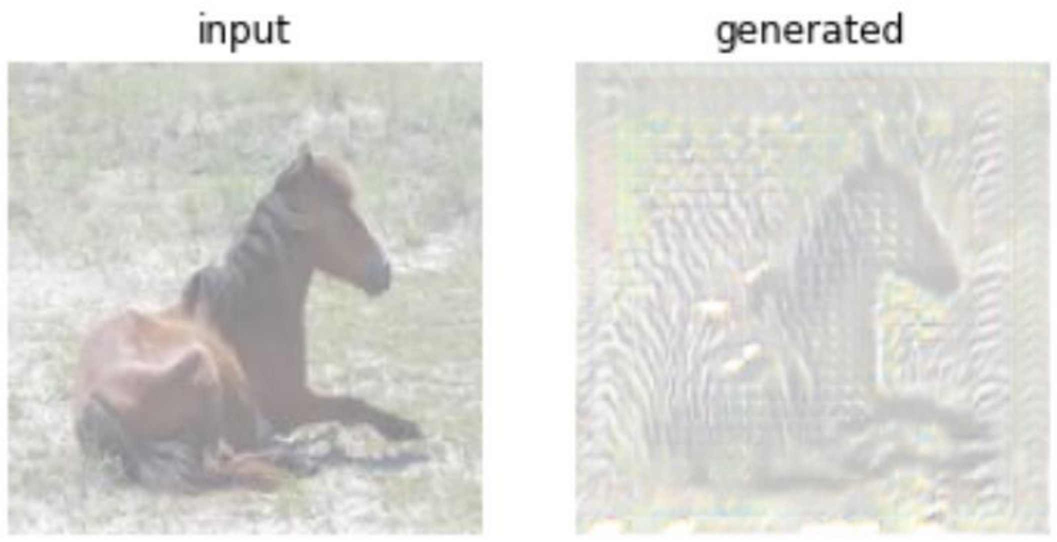


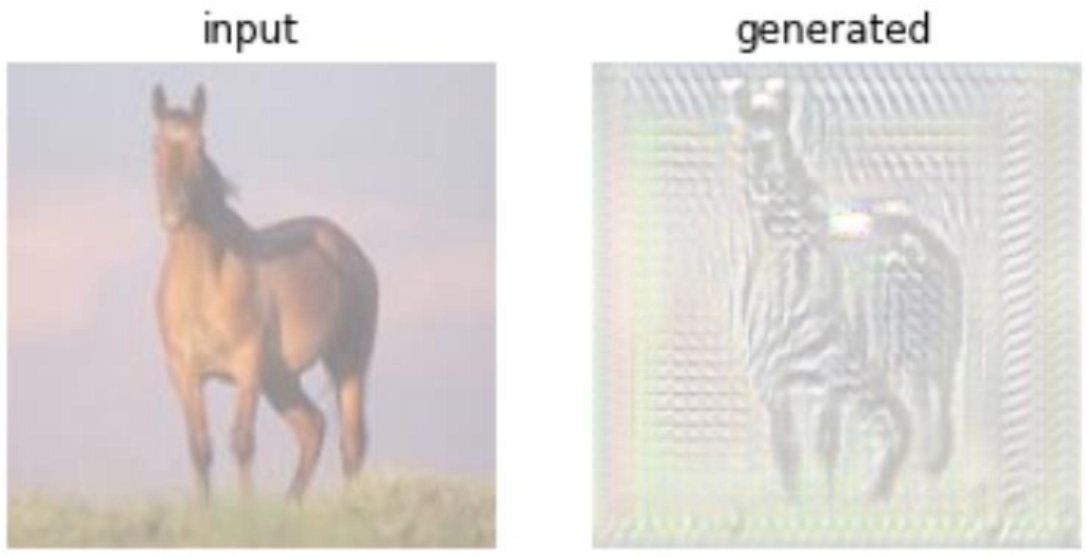
```
Time taken for epoch 268 is 536.287541614532
```

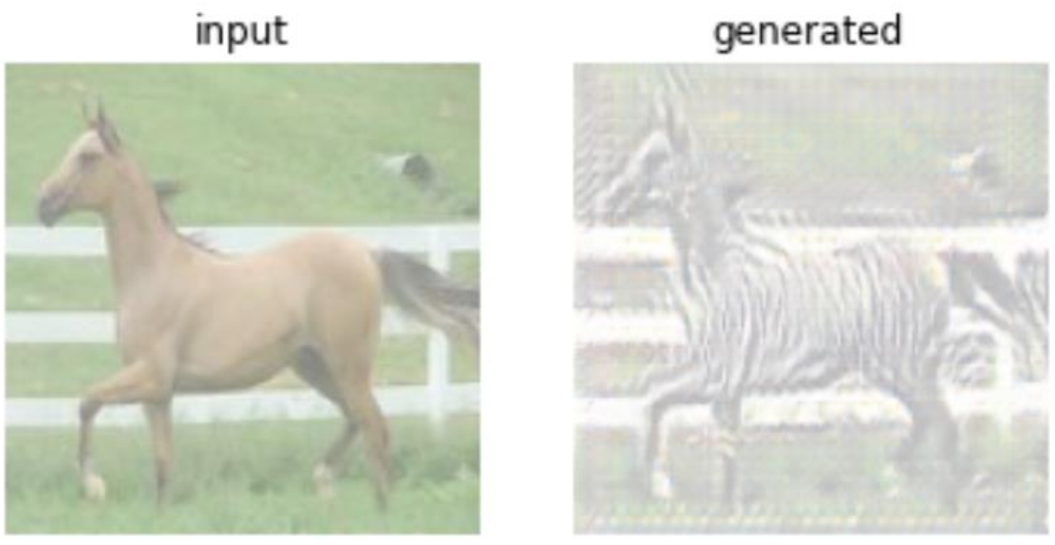


```
Time taken for epoch 252 is 531.9123482704163
```









CONCLUSION

The generator models have generated descent zebra images and overall did good job in the process of conversion.
The project was implemented by the CycleGAN architecture from scratch using the Keras deep learning framework. Specifically, implemented the discriminator and generator models.

DISTRIBUTION OF WORK

Our team has divided work evenly based on each team member's technical background and course load. To be more specific, Naresh worked on pre-processing and testing cycle-GAN models, Jigar worked on building various cycle-GAN models, and Lavy focused on plotting and writing reports.