**1. CycleGAN Architecture**

The CycleGAN architecture consists of two main components: generators and discriminators.

**- Generators (G\_AB and G\_BA):** These networks are responsible for translating images from one domain to another. G\_AB converts images from domain A to domain B, and G\_BA converts images from domain B to domain A. These generators typically use a U-Net-like architecture with encoder and decoder components.

**- Discriminators (D\_A and D\_B):** Discriminators are responsible for distinguishing between real images from their respective domains and fake/generated images. D\_A evaluates images from domain A, while D\_B evaluates images from domain B. Discriminators are typically convolutional neural networks (CNNs).

**2. Implementing the CycleGAN Discriminator Model**

CycleGAN discriminator model using PyTorch:

import torch.nn as nn

class Discriminator(nn.Module):

def \_\_init\_\_(self):

super(Discriminator, self).\_\_init\_\_()

# Define the architecture of the discriminator

self.model = nn.Sequential(

nn.Conv2d(3, 64, kernel\_size=4, stride=2, padding=1),

nn.LeakyReLU(0.2, inplace=True),

# Add more convolutional layers as needed

nn.Conv2d(64, 1, kernel\_size=4, stride=2, padding=1),

nn.Sigmoid()

)

def forward(self, x):

# Forward pass through the discriminator network

return self.model(x)

**3. Implementing the CycleGAN Generator Model**

CycleGAN generator model using PyTorch:

import torch.nn as nn

class Generator(nn.Module):

def \_\_init\_\_(self):

super(Generator, self).\_\_init\_\_()

# Define the architecture of the generator

self.model = nn.Sequential(

# Encoder layers

# Add convolutional and normalization layers here

# Decoder layers

# Add transposed convolutional and normalization layers here

)

def forward(self, x):

# Forward pass through the generator network

return self.model(x)

**The actual architecture of the generator will depend on your specific application and the complexity of the image translation task.**

**4. Implementing Composite Models for Least Squares and Cycle Loss**

To implement composite models for least squares and cycle loss, you can define functions or classes that combine the generators and discriminators to calculate these losses. For example, you can define a function for the least squares loss as follows:

import torch

def least\_squares\_loss(discriminator, real, is\_target\_real):

if is\_target\_real:

target = torch.ones\_like(real)

else:

target = torch.zeros\_like(real)

loss = torch.mean((discriminator(real) - target) \*\* 2)

return loss

Similarly we can define a function or class for the cycle loss, which measures the difference between the input image and the image after a round trip through the generators.

**5. Updating Discriminator and Generator Models**

1. Update Discriminators: Compute the discriminator loss for real and fake images from both domains (D\_A and D\_B). Backpropagate and optimize the discriminators' parameters.

2. Update Generators: Compute the generator loss, which consists of adversarial loss, cycle consistency loss, and optional identity loss. Backpropagate and optimize the generators' parameters.

This alternating update process continues for multiple iterations during training.

PyTorch or TensorFlow are to be used to handle the training process efficiently.