Sketch to Image:

Example 1

Initial Code: Data Augmentation

```
import os
import random
from PIL import Image

def random_scale(image, min_scale=0.8, max_scale=1.2):
    scale_factor = random.uniform(min_scale, max_scale)
    width, height = image.size
    new_width = int(width * scale_factor)
    new_height = int(height * scale_factor)
    scaled_image = image.resize((new_width, new_height), Image.ANTIALIAS)
    return scaled_image

def random_horizontal_flip(image):
    if random.random() > 0.5:
        return image.transpose(Image.FLIP_LEFT_RIGHT)  # Horizontal flip
    return image

def random_vertical_flip(image):
    if random.random() > 0.5:
        return image.transpose(Image.FLIP_TOP_BOTTOM)  # Vertical flip
    return image

input_folder = 'photos'
output_folder = 'augmented_photos'

os.makedirs(output_folder, exist_ok=True)

image_files = [f for f in os.listdir(input_folder)]

for image_file in image_files:
    image_path = os.path.join(input_folder, image_file)
    image = Image.open(image_path)

    for i in range(1000):
        random_h = random_horizontal_flip(image)
        random_v = random_vertical_flip(random_h)
        scaled_image = random_scale(random_v)
        augmented_image_path = os.path.join(output_folder,
f'augmented_image_{i}_{image_file}')
        scaled_image.save(augmented_image_path)

        print(f"Augmented image {i+1}/1000 saved to {augmented_image_path}")
```

Code Review:

- Inefficient way to collect image files without filtering by file extensions
- Lack of proper error handling, like file extension checks
- No comments

Code after Code Review:

```
import os
import random
from PIL import Image

# Function to randomly scale an image
def random_scale(image, min_scale=0.8, max_scale=1.2):
    scale_factor = random.uniform(min_scale, max_scale)
    width, height = image.size
    new_width = int(width * scale_factor)
    new_height = int(height * scale_factor)
    scaled_image = image.resize((new_width, new_height), Image.ANTIALIAS)
    return scaled_image

# Function to randomly flip an image horizontally
def random_horizontal_flip(image):
    if random.random() > 0.5:
        return image.transpose(Image.FLIP_LEFT_RIGHT)  # Horizontal flip
    return image

# Function to randomly flip an image vertically
def random_vertical_flip(image):
    if random.random() > 0.5:
        return image.transpose(Image.FLIP_TOP_BOTTOM)  # Vertical flip
    return image

input_folder = 'photo'
output_folder = 'data_aug_photo'
```

```
os.makedirs(output_folder, exist_ok=True)

image_files = [f for f in os.listdir(input_folder) if f.lower().endswith(('.png', '.jpg', '.jpeg'))]

for image_file in image_files:
    image_path = os.path.join(input_folder, image_file)
    image = Image.open(image_path)

    for i in range(1000):
        random_h = random_horizontal_flip(image)
        random_v = random_vertical_flip(random_h)
        scaled_image = random_scale(random_v)
        augmented_image_path = os.path.join(output_folder, f'aug_{i}_{image_file}')
        scaled_image.save(augmented_image_path)

        print(f"Augmented image {i+1}/1000 saved to {augmented_image_path}")
```

Example 2

Initial Code: Generator

```
class Generator(nn.Module):
    def __init__(self, in_channels=1, out_channels=3, features=[64, 128, 256, 512]):
        super(Generator, self).__init__()

        self.encoder_layers = []
        current_channels = in_channels
        for feature in features:
            encoder_layer = nn.Sequential(
                nn.Conv2d(current_channels, feature, kernel_size=4, stride=2, padding=1),
                nn.BatchNorm2d(feature),
                nn.LeakyReLU(0.2)
            )
            self.encoder_layers.append(encoder_layer)
            current_channels = feature

        self.decoder_layers = []
        reversed_features = list(reversed(features))
        for i in range(len(reversed_features) - 1):
            decoder_layer = nn.Sequential(
                nn.ConvTranspose2d(
                    reversed_features[i] * 2 if i > 0 else reversed_features[i],
                    reversed_features[i + 1],
                    kernel_size=4,
                    stride=2,
                    padding=1
                ),
                nn.BatchNorm2d(reversed_features[i + 1]),
                nn.ReLU()
            )
            self.decoder_layers.append(decoder_layer)

        self.final_transpose_layer = nn.Sequential(
            nn.ConvTranspose2d(
                reversed_features[-1] * 2,
                features[0],
                kernel_size=4,
                stride=2,
                padding=1
            ),
            nn.BatchNorm2d(features[0]),
            nn.ReLU()
        )

        self.final_layer = nn.Conv2d(features[0], out_channels, kernel_size=1)

    def forward(self, x):
        skip_connections = []
        for layer in self.encoder_layers:
            x = layer(x)
            skip_connections.append(x)

        skip_connections = skip_connections[::-1]

        for idx, layer in enumerate(self.decoder_layers):
            x = layer(x)

            if idx < len(skip_connections) - 1:
                skip_feature = skip_connections[idx + 1]
                if x.shape[2:] == skip_feature.shape[2:]:
                    x = torch.cat([x, skip_feature], dim=1)
        x = self.final_transpose_layer(x)
        return self.final_layer(x)
```

Code Review:

- Use nn.ModuleList for encoder and decoder layers instead of Python lists
- Reversing the features list in the decoder initialization is inefficient
- The final_transpose_layer and final_layer can be consolidated into a single module

Code after Code Review:

```
class Generator(nn.Module):
    def __init__(self, in_channels=1, out_channels=3, features=[64, 128, 256, 512]):
        super(Generator, self).__init__()

        self.encoder = nn.ModuleList()
        current_channels = in_channels
        for feature in features:
            self.encoder.append(
                nn.Sequential(
                    nn.Conv2d(current_channels, feature, kernel_size=4, stride=2, padding=1),
                    nn.BatchNorm2d(feature),
                    nn.LeakyReLU(0.2)
                )
            )
            current_channels = feature

        self.decoder = nn.ModuleList()
        reversed_features = list(reversed(features))

        for i in range(len(reversed_features) - 1):
            self.decoder.append(
                nn.Sequential(
                    nn.ConvTranspose2d(
                        reversed_features[i] * 2 if i > 0 else reversed_features[i],
                        reversed_features[i + 1],
                        kernel_size=4,
                        stride=2,
                        padding=1
                    ),
                    nn.BatchNorm2d(reversed_features[i + 1]),
                    nn.ReLU()
                )
            )

        self.final_transpose = nn.Sequential(
            nn.ConvTranspose2d(
                reversed_features[-1] * 2,
                features[0],
                kernel_size=4,
                stride=2,
                padding=1
            ),
            nn.BatchNorm2d(features[0]),
            nn.ReLU()
        )

        self.final_layer = nn.Conv2d(features[0], out_channels, kernel_size=1)

    def forward(self, x):
        skip_connections = []
        for layer in self.encoder:
            x = layer(x)
            skip_connections.append(x)

        skip_connections = skip_connections[::-1]

        for idx, layer in enumerate(self.decoder):
            x = layer(x)

            if idx < len(skip_connections) - 1:
                skip_feature = skip_connections[idx + 1]
                if x.shape[2:] == skip_feature.shape[2:]:
                    x = torch.cat([x, skip_feature], dim=1)

        x = self.final_transpose(x)
        return self.final_layer(x)
```

Example 3

Initial Code: Discriminator

```
class Discriminator(nn.Module):
    def __init__(self, in_channels=4, features=[64, 128, 256, 512]):
        super(Discriminator, self).__init__()
        self.layers = nn.Sequential(
            nn.Conv2d(in_channels, features[0], kernel_size=4, stride=2, padding=1),
            nn.LeakyReLU(0.2),
            nn.Conv2d(features[0], features[1], kernel_size=4, stride=2, padding=1),  #64 to 128
            nn.LeakyReLU(0.2),
            nn.BatchNorm2d(features[1]),
            nn.Conv2d(features[1], features[2], kernel_size=4, stride=2, padding=1),  #128 to 256
            nn.LeakyReLU(0.2),
            nn.Conv2d(features[2], features[3], kernel_size=4, stride=2, padding=1),  #256 to 512
            nn.LeakyReLU(0.2),
            nn.BatchNorm2d(features[3]),
            nn.Conv2d(features[3], 1, kernel_size=4, stride=1, padding=0),
            nn.Sigmoid()  #Ensure output is in the [0, 1] range
        )

    def forward(self, x):
        return self.layers(x)
```

Code Review

- Reordering the BatchNormalization layers after the activation functions might cause a decrease in performance

Discriminator Code after Code Review:

```
class Discriminator(nn.Module):
    def __init__(self, in_channels=4, features=[64, 128, 256, 512]):
        super(Discriminator, self).__init__()
        self.layers = nn.Sequential(
            nn.Conv2d(in_channels, features[0], kernel_size=4, stride=2, padding=1),
            nn.LeakyReLU(0.2),
            nn.Conv2d(features[0], features[1], kernel_size=4, stride=2, padding=1),  #64 to 128
            nn.BatchNorm2d(features[1]),
            nn.LeakyReLU(0.2),
            nn.Conv2d(features[1], features[2], kernel_size=4, stride=2, padding=1),  #128 to 256
            nn.BatchNorm2d(features[2]),
            nn.LeakyReLU(0.2),
            nn.Conv2d(features[2], features[3], kernel_size=4, stride=2, padding=1),  #256 to 512
            nn.BatchNorm2d(features[3]),
            nn.LeakyReLU(0.2),
            nn.Conv2d(features[3], 1, kernel_size=4, stride=1, padding=0),
            nn.Sigmoid()  #Ensure output is in the [0, 1] range
        )

    def forward(self, x):
        return self.layers(x)
```