Université Paris Saclay

Advanced in Machine Vision

Master of Research in Artificial Intelligence and Machine Vision

---

Lab2

Image-to-Image Translation for Semantic Segmentation

Student: Nguyen Trong Dat

Supervisor: Hedi Tabia

Master: M2MMVAI

Academic Year
2023-2024

Contents

List of Figures

Listings

List of Tables

1 Introduction

In this report, we delve into the realm of semantic segmentation using the Cityscape dataset, a benchmark dataset designed for urban scene understanding. The Cityscape dataset provides a diverse collection of high-quality images captured across various cities, making it an invaluable resource for training and evaluating computer vision models, especially in the context of autonomous driving and urban environment analysis.

1.1 Cityscape Dataset: A Brief Overview

The Cityscape dataset comprises images taken from urban street scenes, annotated with pixel-level semantic labels. These annotations include detailed information about objects and regions within the images, such as roadways, vehicles, pedestrians, and more. This dataset serves as a pivotal benchmark for advancing research in semantic segmentation and related computer vision tasks.

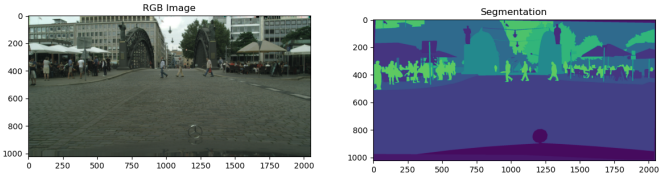
1.1.1 Loading the Dataset

```python
class CityscapeDataset(Dataset):
    def __init__(self, image_path, transform_img=None, transform_label=None):
        self.image_path = image_path
        self.transform_img = transform_img
        self.transform_label = transform_label

    def __len__(self):
        return len(self.image_path)

    def __getitem__(self, idx):
        img = plt.imread(self.image_path[idx])

        image, label = img[:, :img.shape[1] // 2], img[:, img.shape[1] // 2:]

        if self.transform_img:
            image = self.transform_img(image)

        if self.transform_label:
            label = self.transform_label(label)

        return image, label
```

Listing 1: Python code for Loading Dataset

1.1.2 Visualization of Sample Images

Let's visualize one pair of original and ground truth images from the Cityscape dataset to better understand the data. The following figure shows an example:



(a) Original Image (b) Ground truth Image

Figure 1: Cityscape dataset

1.2 Objective of the Study

Our primary objective is to explore and compare the performance of two distinct approaches to semantic segmentation: a conventional model, exemplified by architectures like VAE (Variational

Autoencoder) and U-Net, and a Conditional Generative Adversarial Network (CGAN). Through this comparative analysis, we aim to gain insights into the strengths and limitations of each approach in the specific context of urban scene segmentation.

By leveraging the Cityscape dataset, we aim to assess the models' ability to accurately label and segment different objects and entities within complex urban scenes. The utilization of both traditional and generative adversarial approaches allows us to evaluate the trade-offs between model complexity, computational efficiency, and the quality of segmentation outputs.

This study not only contributes to the ongoing discourse in the field of computer vision but also provides practical insights into the applicability of different models for semantic segmentation in urban environments. The following sections will detail the methodology, experimental setup, results, and conclusions derived from our exploration of these models on the Cityscape dataset.

2 CGAN: Our model

2.1 GAN

A Generative Adversarial Network (GAN) is a class of machine learning models introduced by Goodfellow et al. in 2014. The GAN framework consists of two main components: a generator (G) and a discriminator (D). The primary objective of a GAN is to generate synthetic data that is indistinguishable from real data.

- **Generator (G)**: Takes random noise as input and generates synthetic data.
- **Discriminator (D)**: Takes both real and generated data as input and attempts to distinguish between them.

During training, G aims to create realistic data to fool D, while D improves its ability to differentiate between real and generated samples. This adversarial process results in the generator producing increasingly convincing data over time.

2.2 CGAN: Specifics

A Conditional Generative Adversarial Network (CGAN) extends the GAN framework by introducing conditional information to guide the data generation process. In a CGAN, both G and D receive additional conditional information along with the random noise.

- **Generator (G)**: Takes random noise and conditional information to generate data with specific characteristics.
- **Discriminator (D)**: Receives both real and generated data along with conditional information to make more informed decisions.

The conditional information allows for targeted data generation, making CGANs suitable for tasks where specific attributes or classes are desired in the generated samples.

2.3 Implementation of Conditional GAN (CGAN)

To further explore the application of Conditional Generative Adversarial Networks (CGAN) in the context of semantic segmentation, we present the implementation of a CGAN tailored for our task. The CGAN consists of a generator (G) and a discriminator (D), each designed to accommodate conditional information for targeted image generation.

2.3.1 Generator Implementation

The generator is responsible for synthesizing realistic images based on random noise and conditional information. Below is the Python code for the CGAN generator:

```python
class Generator(nn.Module):
    def __init__(self,ngf, img_size, nc):
        super().__init__()
        self.nc = nc
        self.img_size = img_size
        self.ngf = ngf

        self.downSampling = nn.Sequential(
            nn.Conv2d(nc, self.ngf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(self.ngf, self.ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(self.ngf * 2),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(self.ngf * 2, self.ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(self.ngf * 4),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(self.ngf * 4, self.ngf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(self.ngf * 8),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(self.ngf * 8, self.ngf * 16, 4, 1, 0, bias=False),
        )

        self.upSampling = nn.Sequential(
            nn.ConvTranspose2d( self.ngf * 16, ngf * 16, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 16),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf * 16, ngf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),

            nn.ConvTranspose2d( ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),

            nn.ConvTranspose2d( ngf * 4, ngf*2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf*2),
            nn.ReLU(True),

            nn.ConvTranspose2d( ngf*2, nc, 4, 2, 1, bias=False),
            nn.Tanh()
         )

    def forward(self, input):
        x = self.downSampling(input)
        out = self.upSampling(x)
        return out
```

Listing 2: Python code for Generator

The generator architecture is structured to take into account both the random noise and conditional information, producing images that align with specific characteristics.

2.3.2 Discriminator Implementation

The discriminator evaluates the authenticity of input images, considering both real and generated samples, along with the associated conditional information. Here is the Python code for the CGAN discriminator:

```python
class Discriminator(nn.Module):
    def __init__(self, ngpu, img_size, nc):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.img_size = img_size
        self.nc = nc

```

```
        self.main = nn.Sequential(

            nn.Conv2d(nc*2, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()

        )

    def forward(self, x, labels):
        x = torch.cat([x, labels], 1)
        out = self.main(x)
        return out.view(-1,1,1,1)
```

Listing 3: Python code for Discriminator

The discriminator is designed to make more informed decisions by considering both image content and conditional information.

This CGAN implementation will be utilized in our experiments to assess its effectiveness in semantic segmentation tasks using the Cityscape dataset. In the subsequent sections, we will detail the experimental setup, results, and conclusions derived from applying the CGAN model to our specific use case.

2.4 Training of Conditional GAN (CGAN)

The training loop for the Conditional Generative Adversarial Network (CGAN) involves an adversarial process between the generator and discriminator. The goal is to train the generator to produce realistic images that the discriminator cannot distinguish from real ones. The training code is presented below:

```
print("Starting Training Loop...")

n_of_epochs = 200
loss_Discriminator = 0

for e in range(n_of_epochs):
    running_loss_D = 0.0
    print('Starting epoch {}...'.format(e+1))
    for i, (input,output) in enumerate(train_loader, 0):

        ########################################################################
        ## Update Discriminator: maximize log(D(x/y)) + log(1 - D(G(z/y))) ##
        ########################################################################

        # Loss on real Images
        myDiscriminator.zero_grad()
        predicted = myDiscriminator(input.to(device),output.to(device))
        loss_real = criterion(predicted, torch.ones((len(predicted)), 1, 1,1, dtype
    =torch.float, device=device).to(device))
        loss_real.backward(retain_graph=True)

        # Loss on fake Images
        fake_output = myGenerator(input.to(device))
        predicted = myDiscriminator(input.to(device),fake_output.to(device))
```

```
        loss_fake = criterion(predicted, torch.zeros((len(predicted)), 1, 1,1,
    dtype=torch.float, device=device).to(device))
        loss_fake.backward(retain_graph=True)
        loss_Discriminator = loss_real + loss_fake
        d_optimizer.step()

        ################################################
        ### Update Generator: maximize log(D(G(z/y))) ###
        ################################################

        myGenerator.zero_grad()
        predicted = myDiscriminator(input.to(device),fake_output.to(device))
        loss_gen = criterion(predicted, torch.ones((len(predicted)), 1, 1,1, dtype=
    torch.float, device=device).to(device))
        loss_gen.backward()
        g_optimizer.step()
        running_loss_D +=  loss_Discriminator.item()

    print('g_loss: {}, d_loss: {}'.format(loss_gen, loss_Discriminator))
```

Listing 4: Python code for Training the Model

The training loop consists of two main steps: updating the discriminator to distinguish between real and fake images, and updating the generator to produce more convincing images. These steps are repeated for a specified number of epochs. The loss values for the generator and discriminator are printed for each epoch to monitor the training process.

This training procedure aims to find a balance between the generator and discriminator, resulting in a generator capable of producing synthetic images that align with the conditional information provided.

2.5 Results

As a metric for evaluating the performance of our Conditional Generative Adversarial Network (CGAN) in semantic segmentation, we employ pixel-wise accuracy. Pixel-wise accuracy measures the proportion of correctly classified pixels compared to the total number of pixels in the images.

2.5.1 Pixel-wise Accuracy

Pixel-wise accuracy is calculated using the following formula:

$$PixelwiseAccuracy = \frac{NumberofCorrectlyClassifiedPixels}{TotalNumberofPixels} \times 100 \tag{1}$$

This metric provides insights into how well the model is able to correctly classify each pixel in the generated images.

2.5.2 Results

Dataset	Pixel-wise Accuracy (%)
Training Set	46.41
Validation Set	37.26

Table 1: Pixel-wise Accuracy on Training and Validation Sets

Although these percentages may seem relatively low, visually inspecting the segmented images reveals promising results. The apparent discrepancy between accuracy percentages and visual quality can be attributed to the presence of noise in the generated images.

It's important to note that the model's performance could further improve with additional training epochs. The current results, while demonstrating the potential of the CGAN for semantic segmentation, indicate that more training may lead to a reduction in noise and an enhancement of overall segmentation quality.

2.5.3 Sample Result Image

Let's visualize a sample result image from the validation set:

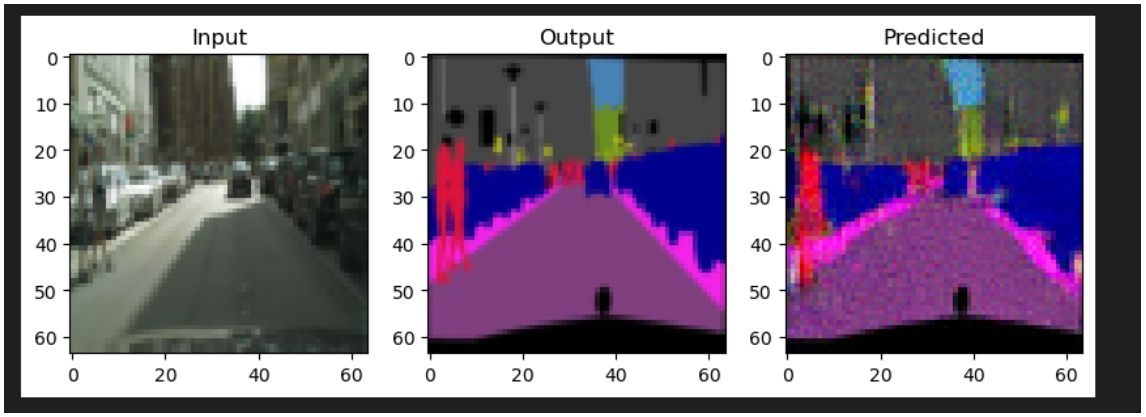


Figure 2: Comparison of Generated Image and Ground Truth

In this example, we can observe the generated image alongside the corresponding ground truth. Despite the pixel-wise accuracy, the visual comparison showcases the model's capability to capture semantic information, providing a foundation for further refinement and improvement.

3 U-Net : Baseline-model

3.1 Introduction to U-Net

U-Net is a convolutional neural network architecture designed for semantic segmentation tasks, introduced by Olaf Ronneberger, Philipp Fischer, and Thomas Brox in 2015. It is particularly well-suited for tasks where precise localization of objects in images is essential, such as medical image segmentation.

The U-Net architecture is characterized by its distinctive U-shaped structure, consisting of an encoder path and a decoder path. The encoder path captures hierarchical features from the input image through a series of convolutional and pooling layers, gradually reducing spatial resolution. The decoder path, on the other hand, upsamples the feature maps and combines them with the corresponding high-resolution features from the encoder, enabling precise localization.

One of the key innovations of U-Net is the introduction of skip connections between the encoder and decoder, allowing the model to retain detailed information during upsampling. This enables U-Net to generate accurate segmentation masks, especially in areas with fine details.

In the context of our study, U-Net serves as the baseline model for semantic segmentation on the Cityscape dataset. Its architecture and proven performance make it a valuable benchmark against which we can compare the results obtained from the Conditional Generative Adversarial Network (CGAN).

3.1.1 Implementation of U-Net

To implement U Net, we define a fundamental building block known as a convolutional block (convblock). This block consists of two convolutional layers with batch normalization and ReLU activation functions. Below is the Python code for the convblock:

```
class conv_block(nn.Module):
    def __init__(self, in_c, out_c):
        super().__init__()
        self.conv1 = nn.Conv2d(in_c, out_c, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(out_c)
        self.conv2 = nn.Conv2d(out_c, out_c, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(out_c)
        self.relu = nn.ReLU(inplace=True)
```

```
    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x = self.relu(x)
        return x
```

Listing 5: Python code for Convolutional Block

The convblock is a crucial component in constructing the encoder and decoder paths of the U-Net architecture. It helps capture and process features at different scales, contributing to the network's ability to perform accurate semantic segmentation.

```
class encoder_block(nn.Module):
    def __init__(self, in_c, out_c):
        super().__init__()
        self.conv = conv_block(in_c, out_c)
        self.pool = nn.MaxPool2d(2, 2)

    def forward(self, x):
        x = self.conv(x)  # for skip connection feature map to decoder
        p = self.pool(x)
        return x, p

class decoder_block(nn.Module):
    def __init__(self, in_c, out_c):
        super().__init__()
        self.convT = nn.ConvTranspose2d(in_c, out_c, kernel_size=2, padding=0,
    stride=2)
        self.conv = conv_block(2 * out_c, out_c)

    def forward(self, inputs, skips):
        x = self.convT(inputs)
        x = torch.cat([x, skips], axis=1)
        x = self.conv(x)
        return x
```

Listing 6: Python code for Encoder and Decoder

The encoderblock handles the downsampling process, consisting of a convolutional block followed by max-pooling. The decoderblock, on the other hand, performs upsampling using transpose convolution and concatenates the features with skip connections from the corresponding encoder block.

These blocks contribute to the overall architecture of U-Net, facilitating the extraction and integration of features at different scales.

To complete the implementation of U-Net, we define the final architecture by combining the previously introduced building blocks. The 'Unet' class serves as the overarching model, consisting of encoder, bottleneck, decoder, and classifier components.

```
class Unet(nn.Module):
    def __init__(self):
        super().__init__()
        """Encoder part"""
        self.en1 = encoder_block(3, 64)
        self.en2 = encoder_block(64, 128)
        self.en3 = encoder_block(128, 256)
        self.en4 = encoder_block(256, 512)

        """Bottleneck"""
        self.b = conv_block(512, 1024)

        """Decoder part"""
        self.d1 = decoder_block(1024, 512)
        self.d2 = decoder_block(512, 256)
        self.d3 = decoder_block(256, 128)
        self.d4 = decoder_block(128, 64)
```

```
        """Classifier"""
        self.outputs = nn.Conv2d(64, 3, kernel_size=1, padding=0)

    def forward(self, input):
        s1, p1 = self.en1(input)
        s2, p2 = self.en2(p1)
        s3, p3 = self.en3(p2)
        s4, p4 = self.en4(p3)

        b = self.b(p4)

        d1 = self.d1(b, s4)
        d2 = self.d2(d1, s3)
        d3 = self.d3(d2, s2)
        d4 = self.d4(d3, s1)

        out = self.outputs(d4)

        return out
```

Listing 7: Python code for Unet

The 'Unet' class orchestrates the entire U-Net architecture, consisting of encoder blocks, a bottleneck, decoder blocks, and a final classifier. This model is designed for semantic segmentation tasks and will be utilized as a baseline for comparison in our study.

3.2 Training U-Net

To train the U-Net model, we follow a standard procedure involving iterative epochs, forward and backward passes, and optimization. The code snippet below exemplifies the training process, where training and validation losses are computed and visualized at regular intervals.

```
train_loss = []
val_loss = []

for epoch in range(epochs):
    trainloss = 0
    valloss = 0
    c = 0

    for img, lab in tqdm(train_loader):
        optimizer.zero_grad()
        img = img.to(device)
        lab = lab.to(device)
        output = model(img)
        loss = loss_func(output, lab)
        loss.backward()
        optimizer.step()
        trainloss += loss.item()

        if epoch % 5 == 0 and c < 2:
            show([img[0], lab[0], output[0]])
            plt.show()
            c += 1

    train_loss.append(trainloss / len(train_loader))

    for img, lab in tqdm(valid_loader):
        img = img.to(device)
        lab = lab.to(device)
        output = model(img)
        loss = loss_func(output, lab)
        valloss += loss.item()

    val_loss.append(valloss / len(valid_loader))

```

```
35      print("epoch: {}, train loss: {}, valid loss: {}".format(epoch, train_loss[-1],
         val_loss[-1]))
```

Listing 8: Python code for Unet

This code utilizes training and validation loaders, computes the loss during training, performs backpropagation, and updates the model's parameters. Additionally, it includes a visualization of sample images at regular intervals.

3.3 Results

3.3.1 Pixel-wise Accuracy

Dataset	Pixel-wise Accuracy (%)
Training Set	56.75
Validation Set	59.88

Table 2: Pixel-wise Accuracy on Training and Validation Sets

3.3.2 Sample Result Image

Let's visualize a sample result image from the validation set:

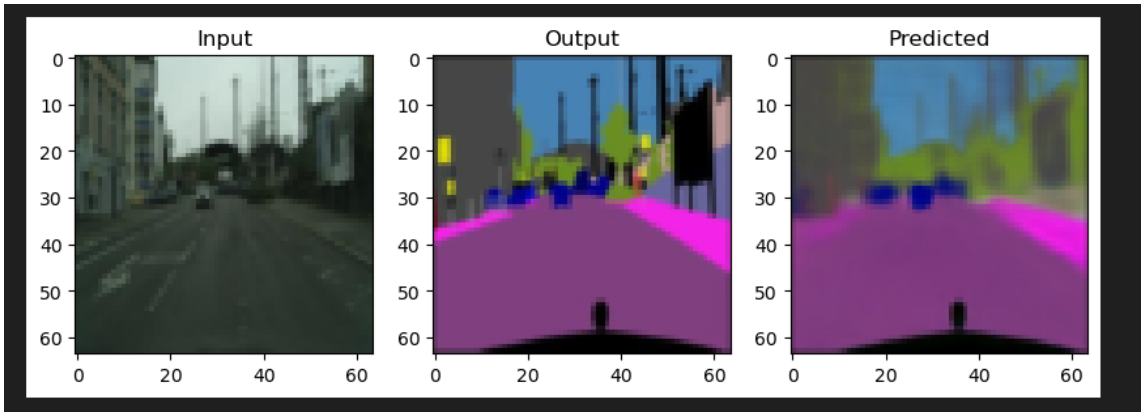


Figure 3: Comparison of Generated Image and Ground Truth (Unet)

In this example, we can observe the generated image using U-Net alongside the corresponding ground truth. The pixel-wise accuracy metrics provide numerical insights, while the visual comparison showcases the model's capability in capturing semantic information.

4 Comparison: U-Net vs. CGAN

4.1 Accuracy Comparison

Model	Pixel-wise Accuracy on Trainset (%)	Pixel-wise Accuracy on Validationset (%)
U-Net	56.75	59.88
CGAN	46.41	37.26

Table 3: Comparison of Pixel-wise Accuracy between U-Net and CGAN

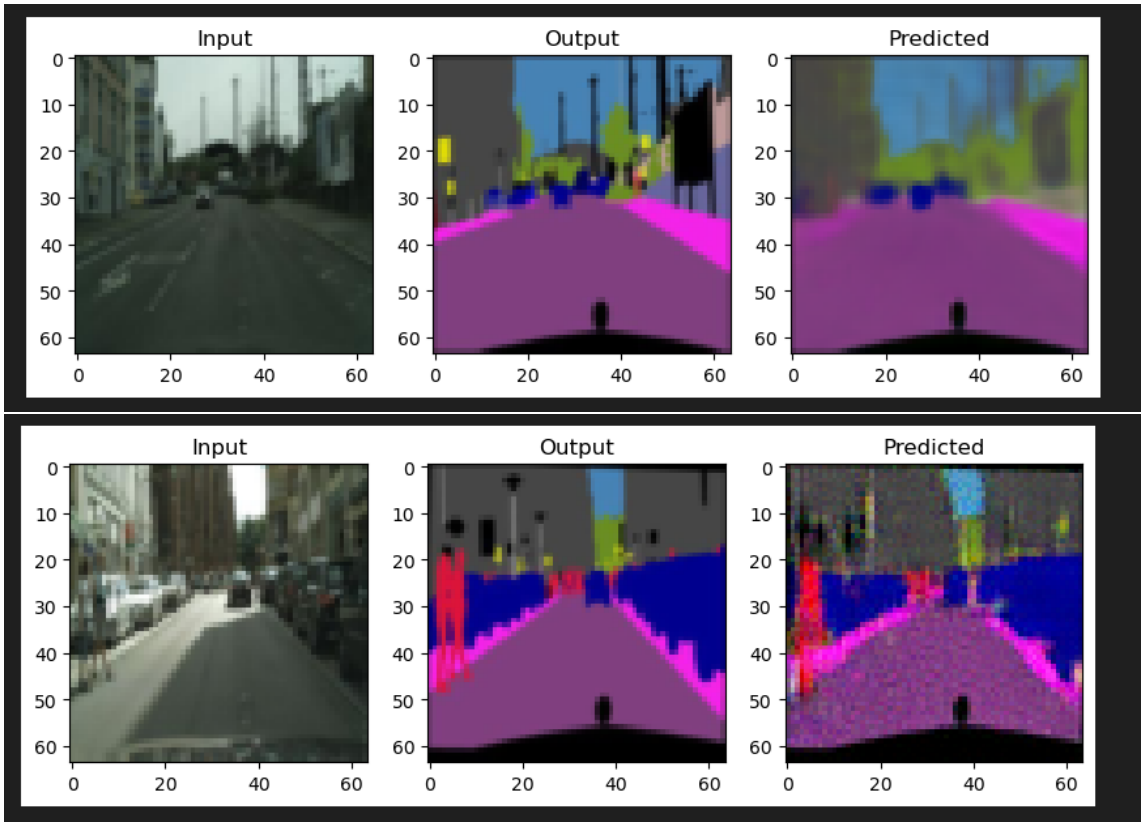


Figure 4: Comparison of Generated Images - U-Net (top) vs CGAN (bottom)

4.2 Generated Images Comparison

4.3 Comment

Even though the pixel-wise accuracy shows that the U-Net gives a better result, the observation reveals that the CGAN produces better-segmented regions with much clearer edges. However, CGAN segmented regions do contain lots of noises, which reduce the accuracy from the pixel-wise accuracy calculation method. Furthermore, the model result still shows to be converging after epochs but slowly. Due to a lack of time and resources, the training of CGAN stopped at 100 epochs. Hence, the CGAN might need more time for training to reach its peak performance.