University of Debrecen
Faculty of Informatics

Simulating Weather Conditions on Digital Images

Supervisor
Andras Hajdu
Professor, PhD

Candidate
Ghais Zaher
Computer Science MSc

Debrecen
2020

Dedication

This dissertation is dedicated to my beloved wife, Hanadi, who encouraged me, stood by my side and supported me as always.

To my parents, my brother and my sister, who raised me and have always been there for me, for their endless love and encouragement.

To my supervisor, Prof. Dr. Andras Hajdu who supported me in both academic and intellectual levels during my Master studies.

Acknowledgements

The code related to this research for both Foggy-CycleGAN and Image Number Annotator projects is implemented by me, Ghais Zaher[*].

Annotations of dataset images is done using Image Number Annotator and performed by me with the help of my wife Hanadi Ebrahim[†].

I would like to pay my special thanks to Mohammad Pouldoust[‡] for his assistance and consultation regarding Deep Learning issues.

All my regards and gratefulness to my supervisor Prof. Dr. Andras Hajdu, who provided me with the needed help and support.

[*]Ghais Zaher, Computer Science MSc, `ghaiszaher@gmail.com`
[†]Hanadi Ebrahim, Software Engineer, `hanadi92ebrahim@gmail.com`
[‡]Mohammad Pouldoust, Machine Learning and Data Mining MSc, `m.pauldoust@gmail.com`

Abstract

Object detection is an essential part of Autonomous Driving and Driver-Assistance systems, the ability to detect and recognize objects in front of the vehicle in addition to its surrounding is the main feedback such systems rely on to take actions. Nevertheless, bad weather conditions like fog, rain and smog decrease the ability to understand the surrounding scene, just like human vision can be impaired in such situations. To overcome this challenge, systems are trained to recognize objects using a dataset of annotated images taken in such weather conditions. One way of building this dataset is to synthesize such phenomena on a set of already segmented images.

Several concerned research use mathematical methods to simulate fog on clear images, benefiting from the physical model of this phenomenon and more analysis of the image content. However, using Convolutional Neural Networks, we can reduce the human effort of finding such 'foggificaion' formulas and let our model figure that out. In this thesis, we propose a generative model that simulates fog on clear images following the idea of CycleGAN [1] and using an unpaired dataset of both foggy and fog-free images. We go through the model formulation process, demonstrate its results and discuss the final outcome.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Numerous weather conditions can impact the driver's vision making it difficult to see some objects on the road, or at least recognize them. Fog is one important phenomenon that can disturb vision while driving [2], scattering the light and making further objects fuzzier. Autonomous Driving systems face the same problem, experiments showed that object detection recall decreases with higher fog density [3]. Several research has been made to detect objects in foggy conditions, either by defogging the image first [4–6], or by applying different object detection methods on the hazy image [7].

Several approaches tried to simulate fog's behaviour and augment it on clear images in order to improve object detection and recognition systems in such weather conditions [8–10], study fog impact on these systems [11, 12] or for other purposes [13, 14], as will be shown in the next section. Most of them focused on the mathematical model that explains how fog is generated in terms of physics, and try to project it on a fog-free photo. Such process is helpful to build up a new dataset of annotated foggy images out of clear ones, which can be used as a training set in an object detection system to increase its accuracy relying on artificial hazy images. Such model will later be used on real photo frames taken during fog existence, which makes it important for the synthesized images to look as much as like real foggy ones as possible, as this would imply better results when real foggy images are fed to the trained model.

In this work, we will present an approach to simulate fog on clear images using a generative model based on GANs [15] and Convolutional Neural Networks. Our main motive is to improve existing Autonomous Driving systems' performance during bad weather conditions with a concentration on fog. Hence, we will focus in our study on images that are mainly taken from a moving vehicle or that contain similar information serving our purpose, in addition to providing more realistic foggy image simulation.

1.2 Related work

1.2.1 Generative Adversarial Networks (GANs)

GANs [15] are generative models that aim to generate data that look similar to the training set. A GAN's structure is demonstrated in Figure 1.1. The input z is a latent vector belonging to the latent space Z, randomly generated and fed to the Generator G, which is responsible for data generation. The discriminator D tries to discriminate between real and fake samples. G is trained to fool D by improving the generated samples to make them look more like real ones, while D is thus getting better at distinguishing the fake samples generated by G from the real ones.

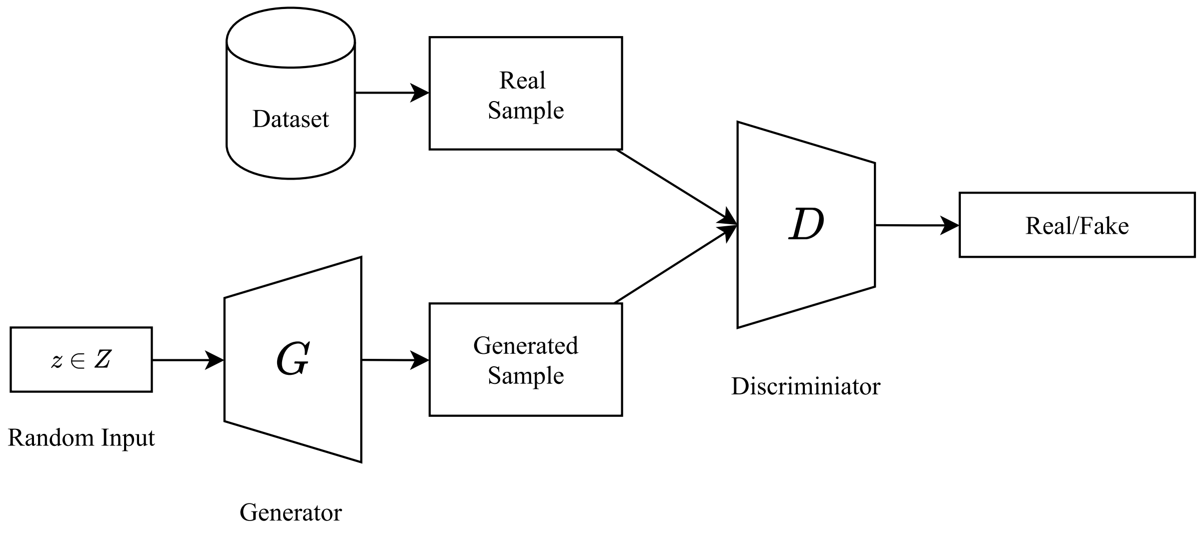


Figure 1.1: GAN Structure

The training demonstrates the following minimax game between G and D:

$$\min_G \max_D V(D, G) = E_x[\log D(x)] + E_z[\log(1 - D(G(z)))]$$

Where:

- E_x is the expected value for real samples
- $D(x)$ is the discriminator's output of the probability that the real instance x is real
- E_z is the expected value for fake samples
- $G(z)$ is the generator's output for input noize z
- $D(G(z))$ is the discriminator's output of the probability that the generated instance $G(z)$ is real

GAN applications focused on image generation using Deep Convolutional neural networks, namely Deep Convolutional GAN or DCGAN [16, 17]. In such applications, the dataset will consist of images that belong to one specific category (e.g. faces, landscapes, digits, etc.). The Generator will learn to generate new images belonging to this category given a random input.

Many improvements have been made to the traditional GAN network. For example, Style-GAN [18, 19] maps the input to an intermediate space, before being fed to the generator at each convolutional layer.

Conditional GAN (CGAN) [20] is another variation of GAN where G and D are provided with a condition y, and the objective function is modified to the following:

$$\min_G \max_D V(D, G) = E_x[\log D(x|y)] + E_z[\log(1 - D(G(z|y)))]$$

We can see from Figure 1.2 that a condition y is provided for every item in the dataset. The discriminator decides whether a sample is real or fake based on the sample itself and the condition. Pix2pix [21] is a good demonstration of CGAN, it tries to generate images based on provided information, for example, generating a handbag image based on an image of its edges, or generating a night photo based on photo taken during daytime.

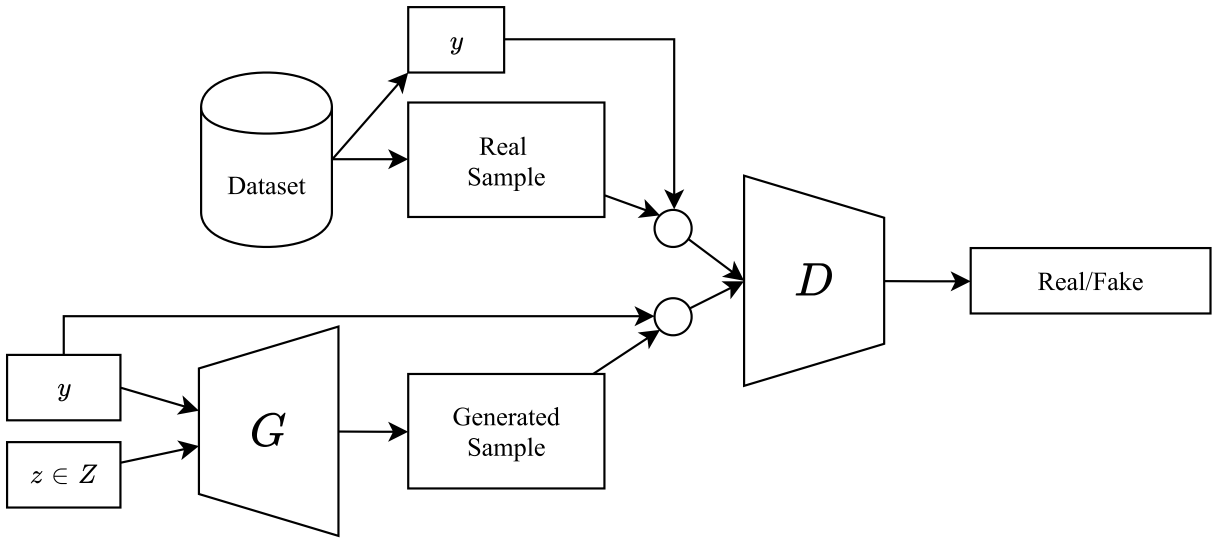


Figure 1.2: CGAN Structure

GANPaint [22] is another application of CGAN, it provides a tool, available online, allowing the user to provide new labels on a part of a given image, and the network would generate a new image modifying the newly labeled parts in a convenient way.

Video generation is another interesting application of GAN, such as vid2vid [23] which provides a model that generates video from a given sequence of video frames, for example, a sequence of semantic segmentation of masks. DVD-GAN (Dual Video Discriminator GAN) [24] and BigGAN [25] introduced models to produce videos on large-scale datasets. Face2Face [26, 27] transmits the facial expressions from one video to another containing a different person in real-time.

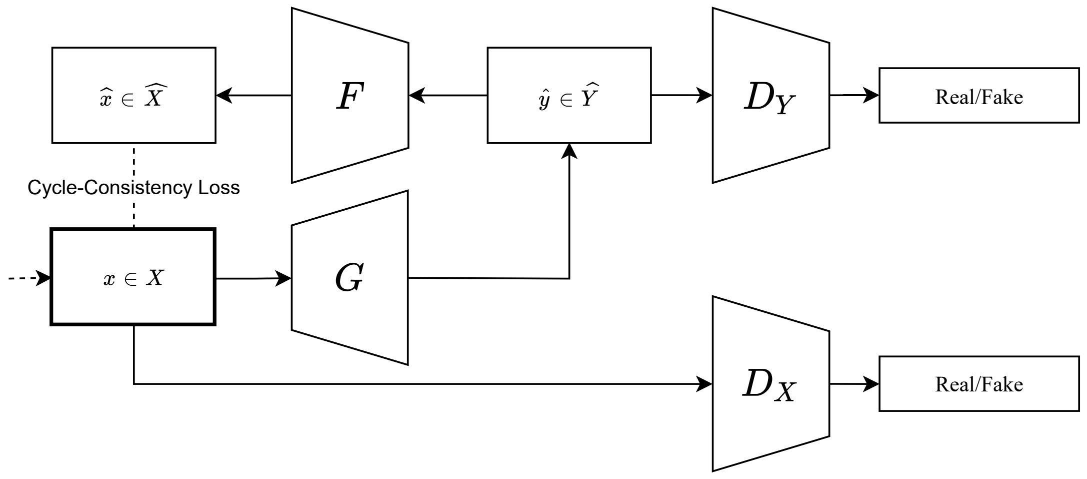
While the main focus of GAN and its variations has been on image and video generation, other applications have been introduced, such as 3D models generation [28] and audio synthesis [29, 30].

1.2.2 Unpaired Image-to-Image Translation

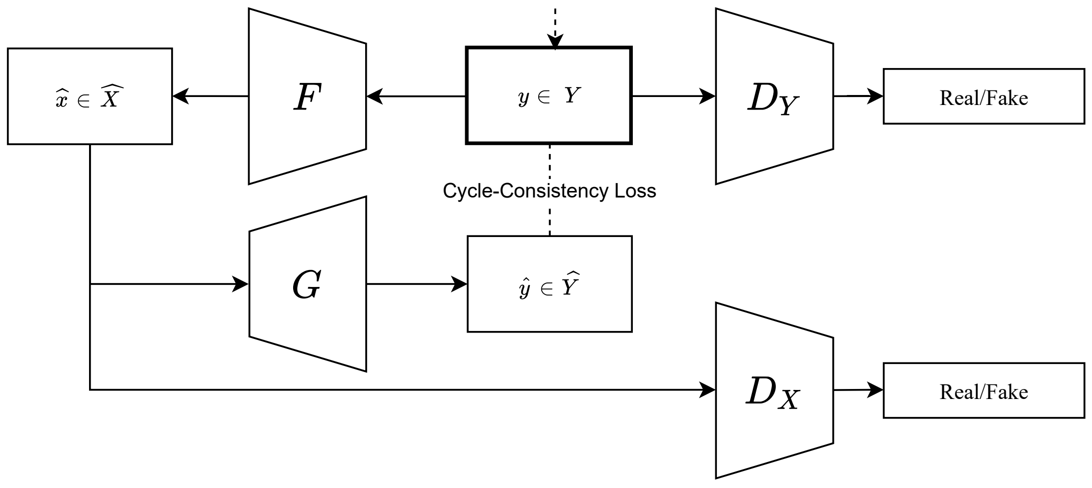
In paired Image-to-Image generation method previously covered [21], every image y in the dataset is paired with another conditional image x, the generator is trained to generate y given the input x. But this requires a lot of efforts to build the dataset. For example, if we want to train a model to convert daylight images to images taken at night, we should have a lot of pairs of photos, each taken from the exact position during day and night time. In other applications, such as ours, it seems to be very difficult to create such real dataset. In order to build a paired training set to generate fog on clear images, we should install static cameras in different places and take pictures during clear and foggy days. This is a very time-consuming method. In addition, the clear photo might contain different objects than the foggy one, and thus we will be training the generator not just to add fog to the images, but also to change its content.

CycleGAN [1] introduced an unpaired Image-to-Image translation method, in which the dataset consists of two sets of images (X,Y), but no relation is defined between an image from the first X set and any of the images from the second Y. CycleGAN tries to imagine what an image from the one collection looks like if it was translated to the other. For example, to know what a photo would look like if it was painted by Monet, we need to train a CycleGAN by providing a set of real photos X, and an unrelated set of Monet paintings Y. The model itself will learn the process and the its reverse, i.e. it would also learn to convert a Monet painting to a real photo.

CycleGAN's structure is illustrated in Figure 1.3, the training process consists of two stages.



(a)



(b)

Figure 1.3: CycleGAN Structure. The model contains two generators $G : X \to Y$ and $F : Y \to X$, and two discriminators D_X and D_Y. D_Y will encourage G to generate outputs that look more like real images from Y, and D_X will encourage F to generate outputs that look more like real images from X.

In the first step shown in Figure 1.3a, an image x from domain X is translated to an image from domain Y by the generator G, the result $\hat{y}$ is converted back to an image $\hat{x}$ from the domain X by the generator F. Both discriminators in this stage are being trained, the fake image $\hat{y}$ is passed to D_Y, and the real one that we started with x is passed to D_X. The final result $\hat{x}$ is supposed to look exactly like x, thus a new loss is introduced here in addition to GAN's adversarial loss previously introduced in subsection 1.2.1 called "Cycle-Consistency Loss":

$$\mathcal{L}_{cyc} = E_x[\|F(G(x)) - x\|_1]$$

The second phase shown in Figure 1.3b is the exact reverse of the first one, starting with an image y from domain Y, converting it to the domain X and back to domain Y, resulting in a "Backward Cycle-Consistency Loss":

$$\mathcal{L}_{backcyc} = E_y[\|G(F(y)) - y\|_1]$$

The complete CycleGAN loss is shown in the following equation:

$$\begin{aligned}\mathcal{L}_{CycleGAN} = {} & E_y[\log D_Y(y)] + E_x[\log D_Y(G(x))] \\ & + E_x[\log D_X(x)] + E_y[\log D_X(F(y))] \\ & + E_x[\|F(G(x)) - x\|_1] + E_y[\|G(F(y)) - y\|_1]\end{aligned} \tag{1.1}$$

The research shows several applications of CycleGAN with intriguing results. For example, converting horses to zebras, apples to oranges, summer to winter and vice-versa.

1.2.3 Fog Synthesis Using Mathematical Models

The basic mathematical model followed in most of fog generation research is Koschmieder's law (atmospheric degradation model):

$$I(x, y) = J(x, y) \cdot t(x, y) + A \cdot (1 - t(x, y)) \tag{1.2}$$

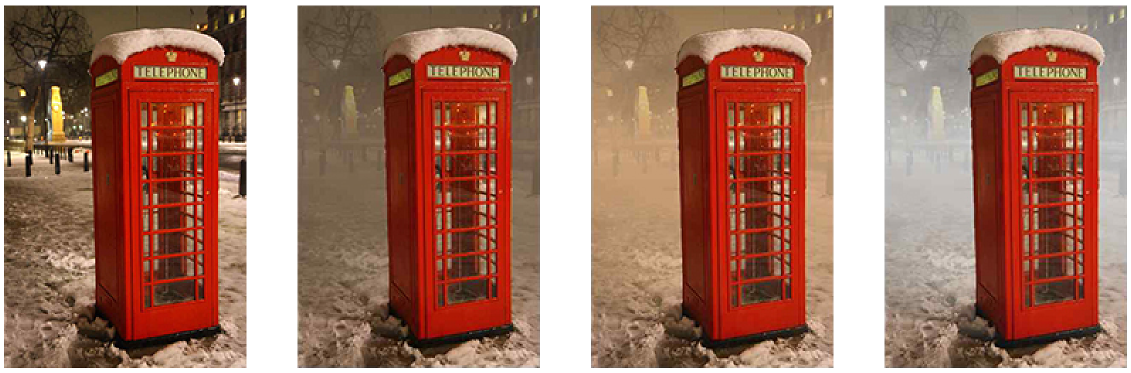
The previous equation is defined on the three RGB channels on a 2D image. (x, y) is an image coordinate. I is the foggy image. J is the fog-free image. A is the atmospheric light, in the day-time, it's mostly considered equal to 255 for each of the RGB colors [8, 13], making the day light equal to (255,255,255), i.e. white. In other cases, this value is estimated from the given clear image. $t(x, y)$ is the transmission map, it expresses the amount of light that survived without being scattered at the coordinate (x, y) of the image. It is related to the depth map by Equation 1.3, where $d(x, y)$ is the distance of the object at coordinate (x, y) and β is the attenuation coefficient that controls fog thickness:

$$t(x, y) = e^{-\beta \cdot d(x,y)} \tag{1.3}$$

β is calculated using Equation 1.4. $C = 3.912$ in [10] and $C = ln(20)$ in [8, 9, 11, 12]. Rm is the maximum visibility distance.

$$\beta = \frac{C}{Rm} \tag{1.4}$$

In order to simulate the foggy image I from the clear one J, we need to estimate the airlight A and the transmission map t, which can be derived from the depth map d. Following this



(a) Original image (b) (120,100,80) (c) (200,160,120) (d) (200,200,200)

Figure 1.4: Fog simulation for different colors [14]

methodology, [14] is estimating a distance-altitude map from a given image using several steps and an interactive graph-cut algorithm, allowing the user to mark certain parts as “objects”. The research is also allowing to change the color of the produced fog, as shown in Figure 1.4.

Same approach is followed in [13], with an additional pseudo-random factor generated as noise, following the idea that fog takes irregular shapes due to wind and air turbulence. In [11], FROSI dataset is built, which contains a set of artificial fog-free images, and for each image, 6 other foggy images with different visibility distances are generated. The research aimed to study the impact of fog presence on traffic sign detection, similar to [12].

Cityscapes dataset [31] is used in [8] to synthesize fog and generate a new dataset “Foggy Cityscapes”, this data was used in the research to train a CNN model for driving assistance. Based on the same model, [9] proposes methods for fog detection, removal and synthesis.

1.2.4 Fog Synthesis Using Generative Models

Research concentrated more on fog removal rather than fog generation. Using generative models and Convolutional Neural Networks, no real work aimed to synthesize fog on clear images, but we will present two papers that proposed fog-removal generative processes that contained a fog generation part.

Cycle-Dehaze [32] introduced a CycleGAN model to deahze a hazy image, but with Cycle-GAN’s structure, adding haze to a clear image was also achieved. A generator G is responsible for dehazing an image, and another generator F produces a hazy image given a clear one.

In addition to CycleGAN’s loss in Equation 1.1, Cycle-Dehaze introduced a “Perceptual Loss” that compares the original image with the final one in feature space rather than pixel space. Considering ϕ as a feature extractor from the $2nd$ and $5th$ layers of VGG16 [33] network, the perceptual loss is defined as follows:

$$\mathcal{L}_{Perceptual} = \|\phi(x) - \phi(F(G(x)))\|_2^2 + \|\phi(y) - \phi(G(F(y)))\|_2^2$$

Cycle-Defog2Refog [34] also uses CycleGAN as a main structure. In the proposed model, the generator `Defog-net` responsible for fog removal out of a foggy image is a straightforward CNN that accepts an image as input and produces a clear image. While `Refog-net` generator that adds fog to the clear image works differently, it relies on the mathematical model (Equation 1.2). `Refog-net` generates the transmission map t from the clear image using a CNN. The atmospheric light A is estimated using a sky prior; the sky region is segmented in the foggy image, then A is equal to the average pixel values of this region, as shown in Equation 1.5. After that, the atmospheric degradation model (Equation 1.2) is applied to generate the foggy image.

$$A_{sky} = \underset{c \in \{r,g,b\}}{mean} (I^c_{sky}(x)) \tag{1.5}$$

After fog removal using `Defog-net` or fog addition using `Refog-net`, the image is processed by `E-D-Net` and `E-R-Net` respectively, these CNNs are used to enhance the generated image removing artifacts and improving its quality.

1.2.5 State-of-the-Art Summary

In summary, several approaches relied on Koschmieder's law (Equation 1.2) to build a mathematical model to synthesize fog on clear images. While few others followed CycleGAN's structure [1] to build a generative model using CNNs that can achieve both fog generation and fog removal, the latter being those methods' original motivation. Even though the mathematical method is based on actual studies on the physical effect of the fog phenomenon, the images generated by such methods seem to be clearly different from real-life foggy images. We believe that a generative model would produce more realistic fog images.

1.3 Outline

In this thesis, we will present the methodology we followed in order to generate fog on clear images. First we will introduce the datasets used in our work, then we will describe our network model including its structure and objective function. After that, we will show the training process of our model. Next, we show and evaluate the results of our method. We end our thesis with a discussion about the contribution we made to fog generation research, the limitations of our proposed method, what future work can be made to improve it, and a final conclusion.

Chapter 2

Methodology

2.1 Introduction

We have seen several methods that are used to synthesize fog on clear images, in our methodology, we are trying to build a generative model, namely “Foggy-CycleGAN”, that has a similar structure to CycleGAN [1] to perform this task. The model aims to add a specific amount (rate) of fog to a clear image, since CycleGAN structure is followed, our model, which consists of two pairs of generators and discriminators, achieves both tasks: simulating fog on clear images and converting foggy images to clear ones.

In order to train our model, we need a dataset of both clear and foggy images. Furthermore, the model’s ability to add a specific amount of fog to a clear image requires our foggy images to have this information. This value will be referred to as the fog ‘Intensity’ in this document. In other research [8] such information is obtained from the Visibility Distance, but for the sake of simplicity, we only provide an estimated percentage of the fog in the image, represented by a number between 0 and 1.

In this chapter, we will cover the dataset preparation process, introduce our Foggy-CycleGAN model, explain its training process and see its results on test images.

2.2 Dataset Preparation

The process of preparing the dataset involved two major steps, collecting both foggy and clear images, then annotating these images with a number that expresses the amount of fog in each of them.

2.2.1 Dataset collection

Several public datasets provide images that can fit our purpose, since our main aim is to improve autonomous driving systems, we focused on images that contain objects that would appear in such systems, e.g. photos taken from a moving car. Cityscapes [31] is a very large dataset, publicly available for researchers, that contains a large number of annotated images taken from moving vehicles in several German cities. In our research, 1526 clear images were collected from this dataset. SFSU Foggy-Driving Dataset [8] provides 101 foggy images from Zurich, 77 of these images were chosen to be part of our dataset. Reside Real-world Task-Driven Testing

Set [35] contains thousands of outdoor images, the majority of them are taken in foggy conditions, 1753 of these images were selected to be part of the dataset. Figure 2.1 shows some samples of the chosen images in each dataset.

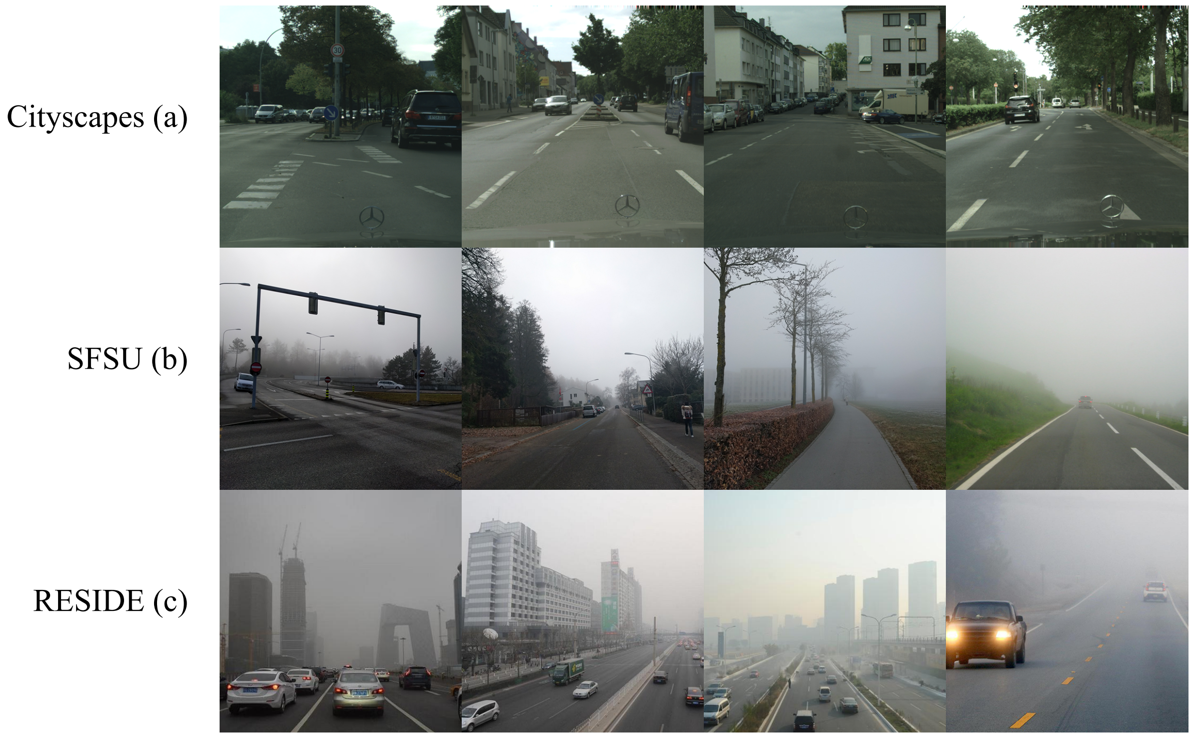


Figure 2.1: Samples of the images in the chosen datasets

2.2.2 Image Number Annotator

The process of annotating our images is to simply assign a number to each image, this number will specify the fog intensity in that image and will be in the range $[0, 1]$, where 0 is assigned to the clear images, 1 is never used since it expresses an image with full fog (i.e. all white image) and no such image exists in the dataset. The previously mentioned datasets originally contained a large number of images, which makes it difficult to annotate them one by one. And since there is no software that can help with such annotation process, I built an application using C# WPF called Image Number Annotator. This application demonstrated in Figure 2.2 allows to easily iterate through all the images in a given directory, and type a number that describes the amount of fog in each observed image.

Image Number Annotator can be used easily, iterating through the images with the keyboard and entering the Intensity immediately, which made the annotation process much faster and easier. The Intensity of an image is manually estimated, there is no exact rule that can determine this number, but it is more of a human estimation of the percentage of fog in the image. Figure 2.3 shows some of these estimations.

Using this tool, the images that we wish to ignore (opt out of the dataset) can be simply skipped and later deleted, the clear images should have a zero intensity, and for the folders that

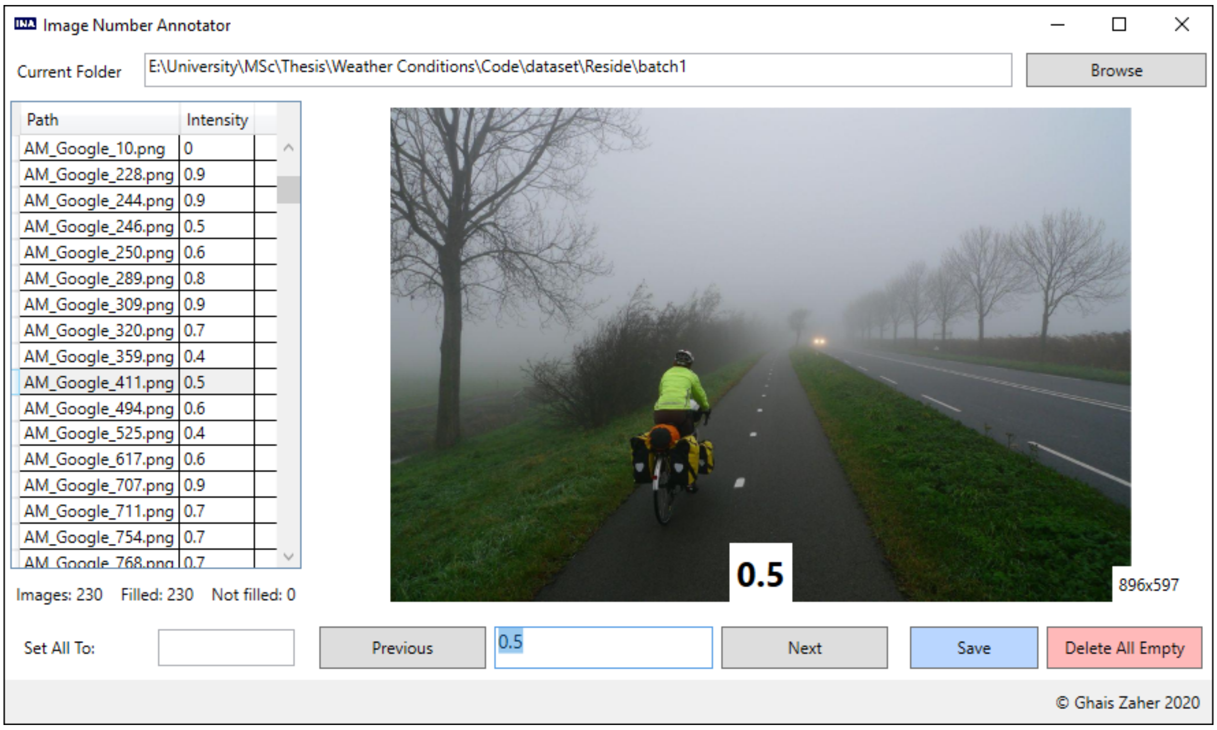


Image Number Annotator

Current Folder E:\University\MSc\Thesis\Weather Conditions\Code\dataset\Reside\batch1

Path	Intensity
AM_Google_10.png	0
AM_Google_228.png	0.9
AM_Google_244.png	0.9
AM_Google_246.png	0.5
AM_Google_250.png	0.6
AM_Google_289.png	0.8
AM_Google_309.png	0.9
AM_Google_320.png	0.7
AM_Google_359.png	0.4
AM_Google_411.png	0.5
AM_Google_494.png	0.6
AM_Google_525.png	0.4
AM_Google_617.png	0.6
AM_Google_707.png	0.9
AM_Google_711.png	0.7
AM_Google_754.png	0.7
AM_Google_768.png	0.7

Images: 230 Filled: 230 Not filled: 0

0.5

896x597

Set All To:

0.5

© Ghais Zaher 2020

Figure 2.2: Image Number Annotator

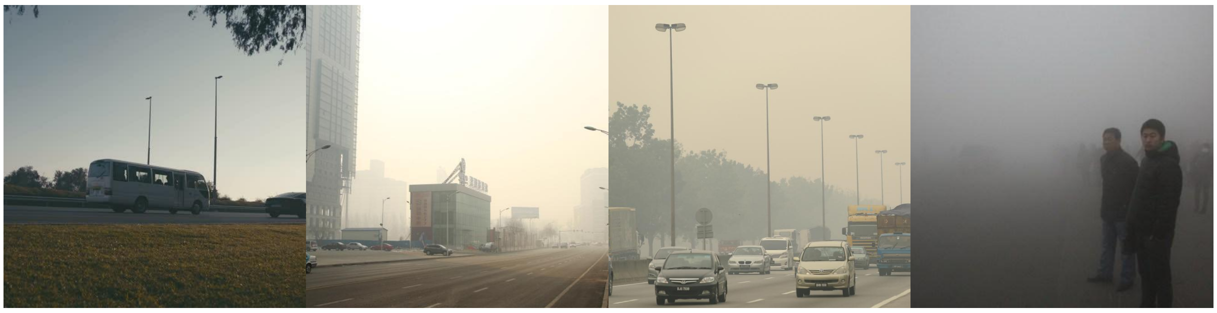

(a) Intensity = 0 (b) Intensity = 0.4 (c) Intensity = 0.6 (d) Intensity = 0.9

Figure 2.3: Manually estimated intensities using Image Number Annotator

contain nothing but clear images, the feature ‘Set All To’ can be used to set all the intensity values to zero.

Finally, the tool will produce a Comma-Separated Values file that will be saved in the same folder that contains the images under the name `Annotations.csv`. This file contains two columns: ‘Path’ which holds the image file name and ‘Intensity’ which contains the estimated intensity. When the process of annotating a folder is done, the button ‘Delete All Empty’ is used to delete all the unannotated images.

2.2.3 Train-Test Split

One clear image and 9 foggy ones were taken aside from the dataset to be part of the “Sample Dataset”, these images were used to observe the training process by plotting the neural networks’ results after each epoch. The rest of the dataset is shuffled then split to 20% test images and 80%

train images, resulting in 2677 images for training (1253 clear and 1424 foggy) and 669 images for testing (313 clear and 356 foggy).

2.3 Foggy-CycleGAN Model Formulation

2.3.1 Model Structure

As introduced in subsection 1.2.2, CycleGAN model consists of two Generators and two Discriminators, so is our model. The essential parts that formulate our structure are:

- $clear2fog$ (Clear to Fog Generator): This generator is provided with a clear image and an intensity value in the range $[0, 1]$. It is responsible for adding the given intensity amount of fog to the clear image. The output of $clear2fog$ is a fake foggy image.

$$clear2fog : Clear \rightarrow Fog$$

- $fog2clear$ (Fog to Clear Generator): This generator is provided with a foggy image and its fog intensity. Its responsibility is to estimate what is behind the fog and replace it with that content, i.e. to remove the fog from the image. The output of $fog2clear$ is a fake clear image.

$$fog2clear : Fog \rightarrow Clear$$

- D_{clear} (Clear Discriminator) and D_{fog} (Fog Discriminator) differ between real and fake images. The discriminators expect an image to be given as an input, a clear image for D_{clear} and a foggy image for D_{fog}. The output of a discriminator is a 30×30 array of real numbers, since this output is only important during the training process, these values were not reduced to one Boolean output, but left as is instead. When a real clear image is given, D_{clear} is expected to output a 30×30 array of ones, otherwise the output should be all zeros. The same applies for D_{clear}.

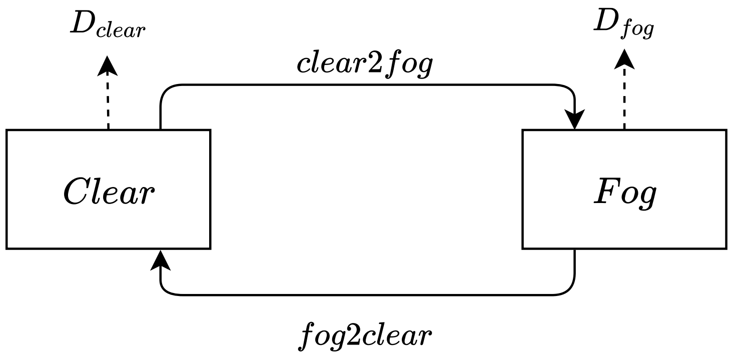


Figure 2.4: Foggy-CycleGAN General Structure

Generators Structure

Both $clear2fog$ and $fog2clear$ share the same final structure. Nevertheless, other implementations of the generator are provided in the final code using `ModelsBuilder`, as will be seen in section 2.3.3. The generators use a modified U-Net [36] structure that has two inputs: a 256×256 colored image with three channels (R,G,B) and a real number that represents the Intensity.

The detailed structure can be seen in Figure 2.5 and Figure 2.6. First, the Intensity value is repeated 256×256 times then added as a fourth layer to the input image, resulting with a 4-channel image (R,G,B,I), where the I represents the intensity. The remaining part of the network is similar to U-Net structure; 8 down-sampling blocks that produce a 1×1 array with 512 filters, then 7 up-sampling blocks that produce a 128×128 array with 128 filters. Finally a De-Convlutional layer is used to up-sample the previous result to a 256×256×3 image. Similar to U-Net, skip connections are used between the down-sampling outputs and concatenated with the equivalent up-sampling outputs as shown in Figure 2.6.

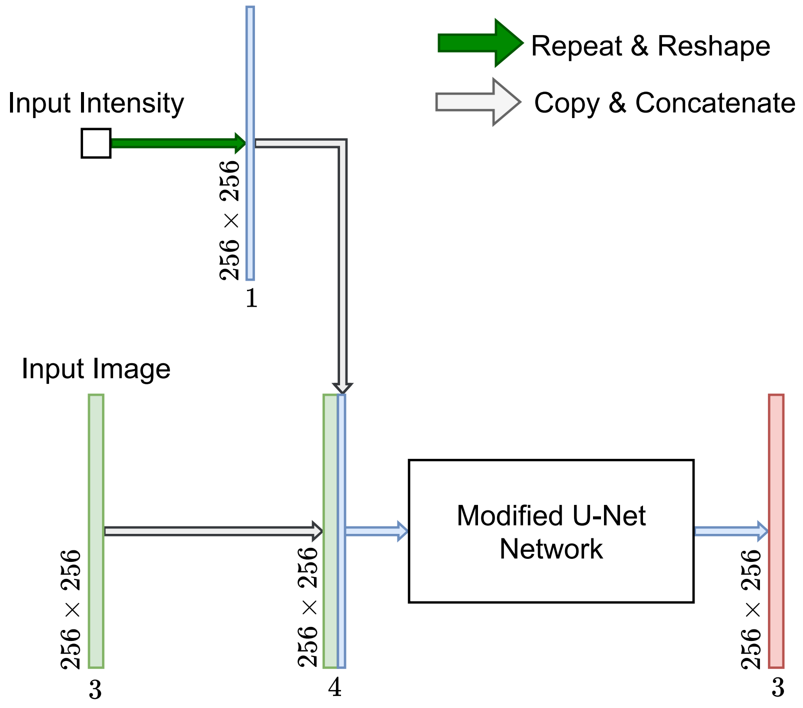


Figure 2.5: Generators Structure: The input intensity is repeated 256×256 times and concatenated to the input image as a fourth channel. The Modified U-Net Network structure is shown in Figure 2.6.

Each down-sampling block consists of the following:

1. A Convolutional layer with a 4×4 kernel size, a stride size of 2×2, same padding and random normal kernel initializer $\sim \mathcal{N}(0, 0.02)$. This results in an output that has half the size of the input. The number of output filters are in order 64, 128 and 256 for the first three down-sampling blocks, then 512 filters for all the rest.

2. Instance Normalization [37] is applied to the filters, except for the first downsample block, as suggested and used by Tensorflow implementation of CycleGAN [38].

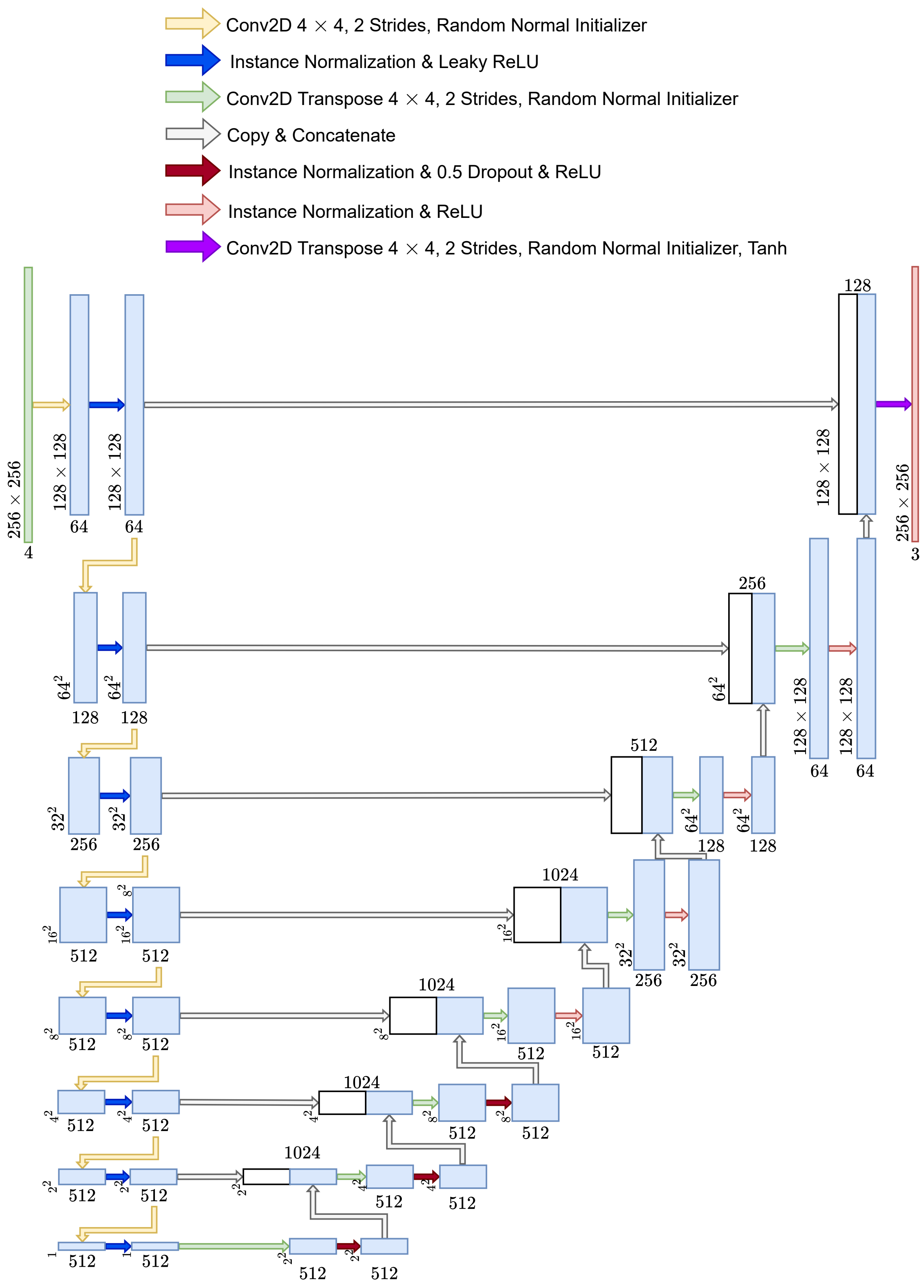

Conv2D 4 × 4, 2 Strides, Random Normal Initializer
Instance Normalization & Leaky ReLU
Conv2D Transpose 4 × 4, 2 Strides, Random Normal Initializer
Copy & Concatenate
Instance Normalization & 0.5 Dropout & ReLU
Instance Normalization & ReLU
Conv2D Transpose 4 × 4, 2 Strides, Random Normal Initializer, Tanh
256 × 256
4
128 × 128
64
128 × 128
64
128
128 × 128
256 × 256
3
64²
128
64²
128
256
64²
128 × 128
64
128 × 128
64
32²
256
32²
256
512
64²
128
64²
128
16²
512
16²
512
1024
16²
32²
256
32²
256
8²
512
8²
512
1024
8²
16²
512
16²
512
4²
512
4²
512
1024
4²
8²
512
8²
512
2²
512
2²
512
1024
2²
4²
512
4²
512
1
512
1
512
2²
512
2²
512

Figure 2.6: Modified U-Net Network Structure: 8 down-sampling blocks followed by 7 up-sampling blocks, skip connections are used between down-sampling and up-sampling block results. Finally, a Deconvolutional layer outputs the final image.

3. Leaky ReLU (Rectified Linear Units) activation with a negative slope of 0.3.

An up-sampling block consists of the following:

1. A Deconvolutional layer with a 4×4 kernel size, a stride size of 2×2, same padding and random normal kernel initializer $\sim \mathcal{N}(0, 0.02)$. This results in an output that has double the size of the input. For each of the first 4 up-sampling blocks, 512 output filters are used. For the last 3 ones, 256, 128 then 64 filters are used (same filters as in down-sampling blocks but in reverse order).

2. Instance Normalization is applied to the filters.

3. Dropout is applied to the first three up-sampling blocks during the training process.

4. ReLU activation.

Every up-sampling block output is concatenated to the output of the corresponding down-sampling block (the one that has the same size and filters). The concatenation is done on the last axis (filters) level as illustrated in Figure 2.6. After the output of the last skip connection between the first down-sampling and the last up-sampling blocks outputs, the result is passed to a Deconvolutional layer with a 4×4 kernel size, a stride size of 2×2, same padding, random normal kernel initializer $\sim \mathcal{N}(0, 0.02)$, 3 filters and hyperbolic tangent activation (`tanh`). The last layer produces an image of 256×256 and 3 channels, the color values range between -1 and 1 (because of the usage of `tanh`).

Resize-Convolution Blocks

Another version of $clear2fog$ (namely $clear2fog$-$v2$) is implemented, where the modified U-Net network uses a different structure for the last 4 up-sampling blocks which we will refer to as Resize-Convolution Blocks. As suggested by [39], instead of using Transposed Convolution for up-sampling, Resize-Convolution resizes the input filters to twice their original size using Bilinear Interpolation, then a standard Convolutional layer is used with same padding and 1×1 strides. A comparison between $clear2fog$ and $clear2fog$-$v2$ and their results will be shown later in section 2.5.

Discriminators Structure

Discriminators D_{clear} and D_{fog} share the same final structure as well. `ModelsBuilder` also allows to do some tweakings to them. The detailed structure of a discriminator shown in Figure 2.7 consists of the following components:

1. The network's input is the image without its intensity value, then three down-sampling blocks produce an image of size 256 filters with a size of 32×32. Those down-sampling blocks are similar to the ones explained in Generators Structure.

2. Zero padding is used to convert the filters sizes to 34×34.

3. A Convolutional layer with a 4×4 kernel size, a stride size of 1×1, no padding, no bias, random normal kernel initializer $\sim \mathcal{N}(0, 0.02)$ and 512 filters.

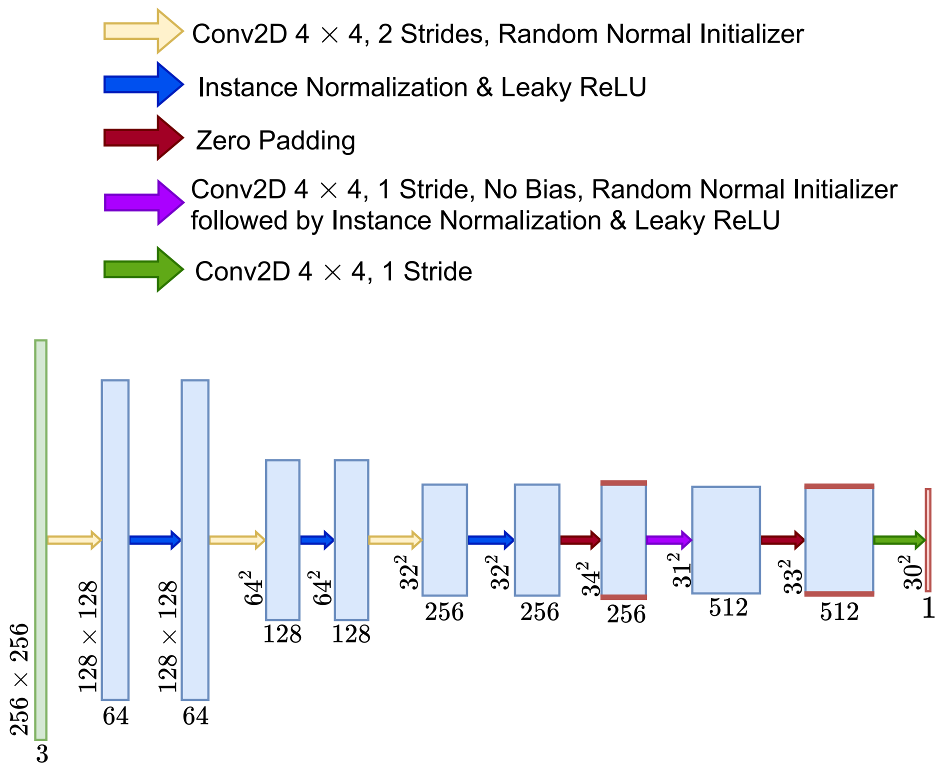

Conv2D 4 × 4, 2 Strides, Random Normal Initializer
Instance Normalization & Leaky ReLU
Zero Padding
Conv2D 4 × 4, 1 Stride, No Bias, Random Normal Initializer followed by Instance Normalization & Leaky ReLU
Conv2D 4 × 4, 1 Stride
256 × 256
3
128 × 128
64
128 × 128
64
64²
128
64²
128
32²
256
32²
256
34²
256
31²
512
33²
512
30²
1

Figure 2.7: Discriminator Structure: The input image is passed to three down-sampling blocks, followed by zero padding and Convolutional layers, resulting with a 30×30 filter.

4. Instance Normalization is applied to the filters.
5. Leaky ReLU activation.
6. Zero Padding
7. A Convolutional layer with a 4×4 kernel size, a stride size of 1×1, no padding, random normal kernel initializer $\sim \mathcal{N}(0, 0.02)$ and one filter. The final output is then one filter of size 30×30.

2.3.2 Losses

Definitions

The following terms will be used to express the losses:

- C: Clear images space.
- F: Foggy images space.
- $I = [0, 1]$: Space of intensity values.
- E_f: the expected value for foggy samples.
- E_c: the expected value for clear samples.

- $f \in F$ is a foggy image and $i_f \in I$ is its intensity.
- $c \in C$ is a clear image and $i_c \in I$ is its intensity.

Adversarial Loss

Similar to the standard one introduced in [15]:

$$\mathcal{L}_{\text{GAN}}(clear2fog, D_{fog}, C, F, I) = E_f[\log D_{fog}(f, i_f)] + E_c[\log(1 - D_{fog}(clear2fog(c, i_c), i_c))]$$

$$\mathcal{L}_{\text{GAN}}(fog2clear, D_{clear}, F, C, I) = E_c[\log D_{clear}(c)] + E_f[\log(1 - D_{clear}(fog2clear(f, i_f)))]$$

Cycle-Consistency Loss

As introduced in [1]:

$$\mathcal{L}_{cyc} = E_c[\|fog2clear(clear2fog(c, i_c), i_c) - c\|_1]$$

$$\mathcal{L}_{backcyc} = E_f[\|clear2fog(fog2clear(f, i_f), i_f) - f\|_1]$$

Identity Loss

Passing a foggy image to $clear2fog$ and passing a clear image to $fog2clear$ should always generate the same image:

$$\mathcal{L}_{identity} = E_f[\|clear2fog(f, i_f) - f\|_1] + E_c[\|fog2clear(c, i_c) - c\|_1]$$

Transmission Map Loss

This loss makes sure that converting a clear image to a foggy one with a required intensity value i_c will produce a new image $clear2fog(c, i_c)$ whose average color should be similar to the original one c after adding i_c amount of whitening to the pixels.

This is called Transmission Map loss because it benefits from the mathematical model in Equation 1.2. Considering $A = 1$, the model suggests that a pixel (x, y) is whitened by the amount of $1 - t(x, y)$, if this last amount is equal to 1, the pixel will become full white, if it is zero then it will keep its original color.

In order to keep our ‘foggification’ balanced along different intensities between 0 and 1, we use the mathematical model but only on the mean color of the image. Ergo, the mean color of $clear2fog(c, i_c)$ should be equal to the mean color of c after whitening each pixel with an amount of i_c.

$$\mathcal{L}_{trans1} = E_c|mean(clear2fog(c, i_c)) - mean(c \cdot (1 - i_c) + i_c)|$$

In addition to that, passing an intensity value of zero should produce the same image and passing the intensity value of 1 should produce a full-white image:

$$\mathcal{L}_{trans2} = E_c[\|clear2fog(c, 0) - c\|_1 + \|clear2fog(c, 1) - w\|_1]$$

Where w is a full-white image.

The full Transmission Map loss is the sum of these 2 losses:

$$\mathcal{L}_{trans} = \mathcal{L}_{trans1} + \mathcal{L}_{trans2}$$

Whitening Loss

Whitening loss along with RGB Ratio loss, make sure that any pixel in the generated fog image is either whiter or the same as the original. This loss ensures that the image's color is not changed towards a darker one, it is only whitened. This is done by using the ReLU function (ReLU$(x) =$ max$(x, 0)$):

$$\mathcal{L}_{white} = E_c[\|\text{ReLU}(c - clear2fog(c, i_c))\|_1]$$

RGB Ratio Loss

Previous loss makes sure that every channel of the RGB colors is larger than the original, but if any of them is increased more than the others, the color will not look whiter, it will be distorted. RGB Ratio loss makes sure that R,G and B colors are being increased with the same ratio:

$$\mathcal{L}_{rgb} = E_c[\|(r \cdot \hat{g} - g \cdot \hat{r})\|_1 + \|(g \cdot \hat{b} - b \cdot \hat{g})\|_1]$$

Where r, g, b are the channels of the input image c and $\hat{r}, \hat{g}, \hat{b}$ are the channels of the generated image $clear2fog(c, i_c)$.

Full Objective

The full objective function to be minimized:

$$\begin{aligned} \mathcal{L}_{Foggy\text{-}CycleGAN} &= \mathcal{L}_{\text{GAN}}(clear2fog, D_{fog}, C, F, I) \\ &+ \mathcal{L}_{\text{GAN}}(fog2clear, D_{clear}, F, C, I) \\ &+ \lambda\mathcal{L}_{cyc} + \lambda\mathcal{L}_{backcyc} + \lambda\mathcal{L}_{identity} \\ &+ \lambda\mathcal{L}_{trans} + \alpha\lambda\mathcal{L}_{white} + \alpha\lambda\mathcal{L}_{rgb} \end{aligned} \tag{2.1}$$

α is the multiplier rate that gives higher importance and thus higher punishment to the neural network for these losses. In our model $\alpha = 10$ and $\lambda = 5$.

2.3.3 Implementation

The implementation was essentially done using Python 3, Tensorflow 2.1.0 and Numpy. Additionally, the following libraries were used:

- Matplotlib library is used for visualization.
- Jupyter is used to write a Notebook where the model is trained and tested.
- Pydot and Graphviz to draw models' structures.

- Pandas is used to read the annotation files produced by Image Number Annotator introduced in subsection 2.2.2 and prepare the train and test datasets.

The code is divided to multiple parts to make it dynamic to create, change, train and test the models.

Dataset Initializer

The class `DatasetInitializer` in `dataset.py` is responsible for preparing the dataset. Essentially, the function `prepare_dataset` does all the important work by calling other methods. Generally, the following steps are done by `prepare_dataset`:

1. Reading the annotation files recursively from the dataset folder, by searching for all files named `Annotations.csv`. The files are read using Pandas into one `DataFrame`, where each row contains the full path of an image and its fog intensity.
2. The dataframe is split into two: clear images that have zero intensity and foggy images that have positive intensity.
3. Each of the two dataframes is shuffled and split into train and test parts, as specified in subsection 2.2.3.
4. An additional `DataFrame` is set similarly for the Sample images, those images consist of one clear image, and 9 foggy ones with intensities between 0.1 and 0.9.
5. In the end, the following generators are returned:
    - train clear generator: shuffles the training clear dataframe, then iterates through its rows one by one
    - train fog generator: shuffles the training foggy dataframe, then iterates through its rows one by one
    - test clear generator: shuffles the testing clear dataframe, then iterates through its rows one by one
    - test fog generator: shuffles the testing foggy dataframe, then iterates through its rows one by one
    - sample clear generator: iterates through 9 different intensities for the sample clear images, ranging from 0.1 to 0.9.
    - sample fog generator: iterates through the sample foggy images.
6. Each of the previous generators returns a row of an image path and an intensity value. Images are then read from their path and normalized as follows:
    - Image is read from the disk.
    - Since the generators use `tanh` activation, all the images are normalized to the range $[-1, 1]$.
    - Image is resized to 256×256, this is done with another method by maintaining the image's aspect ratio.
    - Jitter is applied to the training images by resizing them up to 286×286 then cropped randomly back to 256×256. The image is also randomly flipped.

Models Builder

The class ModelsBuilder is used to build Generators and Discriminators in a generic way, all the neural networks and layers are implemented using Tensorflow and Keras. The following methods are the essential parts of the class explained in subsection 2.3.1:

- downsample: Returns a Down-sampling block with the passed number of filters and kernel size.
- upsample: Returns an Up-sampling block with the passed number of filters, kernel size and whether Dropout is applied or not.
- resize_conv: Returns a Resize-Convolution block with the passed number of filters, kernel size and whether Dropout is applied or not.
- build_generator: This function takes the parameters that define a generator and builds up one.
- build_discriminator: This function takes up the parameters that define a discriminator and builds up one.

Trainer

Trainer class in train.py is responsible for the whole training process for the Foggy-CycleGAN model. Loss functions, weights update, saving checkpoints and plotting results are all handled and defined in this class. Trainer uses Adam optimizer [40] to train all the models with a learning rate of 10^{-4}, $\beta_1 = 0.5$ and $\beta_2 = 0.999$. The class provides the following methods:

- save_config: Saves the current configuration of the Trainer, from weights path, total trained epochs and logs paths.
- load_config: Loads the configuration into the class.
- configure_checkpoint: Loads the weights of the models.
- save_weights: Saves the current weights of the models.
- train_step: This function is responsible for one training step, taking one batch of clear images, and one batch of foggy ones. The function calculates all the losses explained in subsection 2.3.2, calculates the gradient and updates the weights with the help of Tensorflow.
- epoch_callback: This method is called after every epoch, it plots the result of a random sample clear and sample fog pair, showing the generators and discriminators results on them. In addition, it stores such prediction for all the sample images in a specified folder for logging.
- train: This is the main method for the training, it calls several other methods in order to perform a given number of epochs. In addition it is responsible for storing Tensorboard logs.

In train_step, the passed parameter real_clear contains a tuple of clear images along with random intensities for each image. And the parameter real_fog contains a tuple of foggy images along with their fog intensity. The losses are calcualted as follows:

- real_clear is converted to fog using $clear2fog$, and real_clear is converted to clear using $fog2clear$.

```
fake_fog = generator_clear2fog((real_clear, clear_intensity))
fake_clear = generator_fog2clear((real_fog, fog_intensity))
```

- real_fog and fake_fog are passed to D_{fog}. real_clear and fake_clear are passed to D_{clear}. The results are used to calculate **Adversarial Loss**.

```
disc_real_clear = discriminator_clear(real_clear)
disc_real_fog = discriminator_fog((real_fog, fog_intensity))
disc_fake_clear = discriminator_clear(fake_clear)
disc_fake_fog = discriminator_fog((fake_fog, clear_intensity))
disc_clear_loss = discriminator_loss(disc_real_clear, disc_fake_clear)
disc_fog_loss = discriminator_loss(disc_real_fog, disc_fake_fog)
```

  Where discriminator_loss calculates the binary crossentropy value between the real image and an array of ones, and between the fake image and an array of zeros.

- fake_fog image is converted back to clear and fake_clear is converted back to fog to calculate **Cycle-Consistency Loss**.

```
cycled_clear = generator_fog2clear((fake_fog, clear_intensity))
cycled_fog = generator_clear2fog((fake_clear, fog_intensity))
total_cycle_loss = LAMBDA *
   tf.reduce_mean(tf.abs(real_clear-cycled_clear)) + LAMBDA *
   tf.reduce_mean(tf.abs(real_fog-cycled_fog))
```

- **Identity Loss** is calculated by passing the foggy image to $clear2fog$ and the clear one to $fog2clear$ and expecting the same result.

```
same_clear = generator_fog2clear((real_clear, clear_intensity))
same_fog = generator_clear2fog((real_fog, fog_intensity))
identity_loss = LAMBDA * tf.reduce_mean(tf.abs(real_image - same_image))
```

- **Transmission Map Loss** is calculated by generating a transmission image using the mathematical formula.

```
t = 1 - intensity
trans_image = clear_image * t + (1 - t)
trans_loss1 = LAMBDA * tf.abs(tf.reduce_mean(real_image) -
   tf.reduce_mean(trans_image))
```

  The second part is calculated by generating a foggy image from the clear one with both 0 and 1 intensities, the result should be the same image and full white respectively.

```
fake_clear2clear = generator_clear2fog((real_clear, 0))
fake_clear2white = generator_clear2fog((real_clear, 1))
white = tf.ones_like(real_clear)
trans_loss2 = LAMBDA * tf.reduce_mean(tf.abs(real_clear -
    fake_clear2clear)) + LAMBDA * tf.reduce_mean(tf.abs(white -
    fake_clear2white))
```

- **Whitening Loss** calculation is straightforward.

```
whitening_loss = LAMBDA * ALPHA * tf.reduce_mean(tf.maximum(real_clear -
    fake_fog, 0))
```

- For **RGB Ratio Loss**, red, green and blue channels are extracted from both clear and generated images, then the loss is calculated.

```
r = clear_image[:, :, :, 0]
g = clear_image[:, :, :, 1]
b = clear_image[:, :, :, 2]
r_hat = fake_fog[:, :, :, 0]
g_hat = fake_fog[:, :, :, 1]
b_hat = fake_fog[:, :, :, 1]
rg_loss = tf.reduce_mean(tf.abs(r * g_hat - g * r_hat))
gb_loss = tf.reduce_mean(tf.abs(g * b_hat - b * g_hat))
rgb_loss = LAMBDA * ALPHA * (rg_loss + gb_loss)
```

Tools and Plot Libraries

The file tools.py contains some helping methods like printing with timestamps for logging creating directories and returning a dataframe length in an optimal way. The library plot.py has the methods responsible for plotting the input images, the generated ones and the discriminators outputs. Such function are separated in their own modules because they are needed in several other modules and classes.

Main Notebook

The Jupyter Notebook file Foggy_CycleGAN.ipynb holds the main part that uses the previous modules and classes to prepare the dataset, build the generators and discriminators, load pre-saved models if any, set up Tensorboard, start the training process and plot the results.

The dataset is prepared using DatasetInitializer, as shown in Listing 2.1, the parameters passed to DatasetInitializer are the image height and width, in our case the image size is 256×256. BATCH_SIZE constant represents the batch size for the training process, in our case its value is 5.

Listing 2.1: Preparing the dataset

```
datasetInit = DatasetInitializer(256, 256)
```

```
(train_clear, train_fog), (test_clear, test_fog), (sample_clear, sample_fog) =
    datasetInit.prepare_dataset(BATCH_SIZE, test_split=0.2, random_seed=7)
```

Generators and discriminators are built using `ModelsBuilder`, as demonstrated in Listing 2.2. The parameter `use_resize_conv` specifies whether the generator is built using Resize Convolutional layers instead of regular up-sampling blocks. In *clear2fog* we are using regular up-sampling blocks passing this value as `False`, while in *clear2fog-v2* it is passed as `True`.

Listing 2.2: Building the generators and discriminators

```
models_builder = ModelsBuilder()
generator_clear2fog = models_builder.build_generator(use_resize_conv=False)
generator_fog2clear = models_builder.build_generator()
discriminator_fog = models_builder.build_discriminator()
discriminator_clear = models_builder.build_discriminator()
```

Training process starts by initializing the `Trainer` and configuring its checkpoints which loads the weights if they already exist, as show in Listing 2.3. Calling `load_config()` will load the last saved configuration for the trainer, which contains the number of trained epochs, the paths for saving weights and Tensorboard logs.

Listing 2.3: Initialize Trainer

```
trainer = Trainer(generator_clear2fog, generator_fog2clear, discriminator_fog,
    discriminator_clear)
trainer.configure_checkpoint(weights_path = weights_path,
    load_optimizers=False)
trainer.load_config()
```

The training process starts by calling `trainer.train` function, `clear_output_callback` is used to clear the notebook output after each epoch, `use_tensorboard` tells the trainer that tensorboard logs need to be saved, `sample_test` is used to plot images after each epoch, as well as storing them in a folder to visualize them later in section 2.4 and section 2.5

Listing 2.4: Start Training Process

```
trainer.train(
    train_clear, train_fog,
    epochs=100,
    clear_output_callback=lambda: clear_output(wait=True),
    use_tensorboard = True,
    sample_test =(sample_clear, sample_fog),
    load_config_first=False,
    use_whitening_loss=True,
    use_rgb_ratio_loss=True
)
```

Code Availability

The code is publicly available in my GitHub repository github.com/ghaiszaher/Foggy-CycleGAN.

2.4 Training

2.4.1 Environment Setup

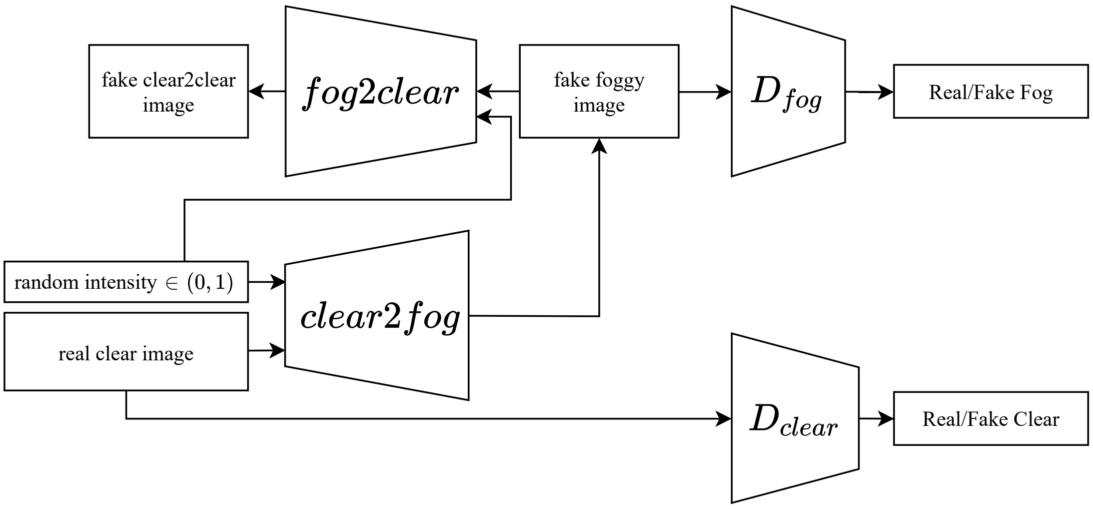
GitHub and Google Colaboratory (Colab) are mainly used to host the code and the training process. The code is pushed to a GitHub repository, and the main Notebook `Foggy_CycleGAN.ipynb` is run on Colab. Google Colaboratory allows us to create and run Jupyter Notebook files on a Linux virtual machine, the option to run the code on CPU, GPU or TPU (Tensor Processing Unit) with limited resources and the ability to mount a Google Drive account to the machine. For this purpose, the dataset files are zipped and uploaded to a Google Drive folder.
The following process happens when the Notebook file is run on Colab:

1. Runtime is set to use GPU resources.
2. Code is pulled from Github
3. Google Drive is mounted to Colab.
4. A shell script `copy_dataset.sh` is run to copy the compressed dataset files to the machine and decompress them.
5. Dataset is initialized using `DatasetInitializer` with a batch size of 5 images.
6. Generators and Discriminators are built using `ModelsBuilder`
7. Tensorboard is initialized locally and shown in the same Notebook
8. The model is trained for 100 epochs at a time. Weights and logs are all stored in the mounted Google Drive.
9. When the training is done, testing results are visualized and stored.

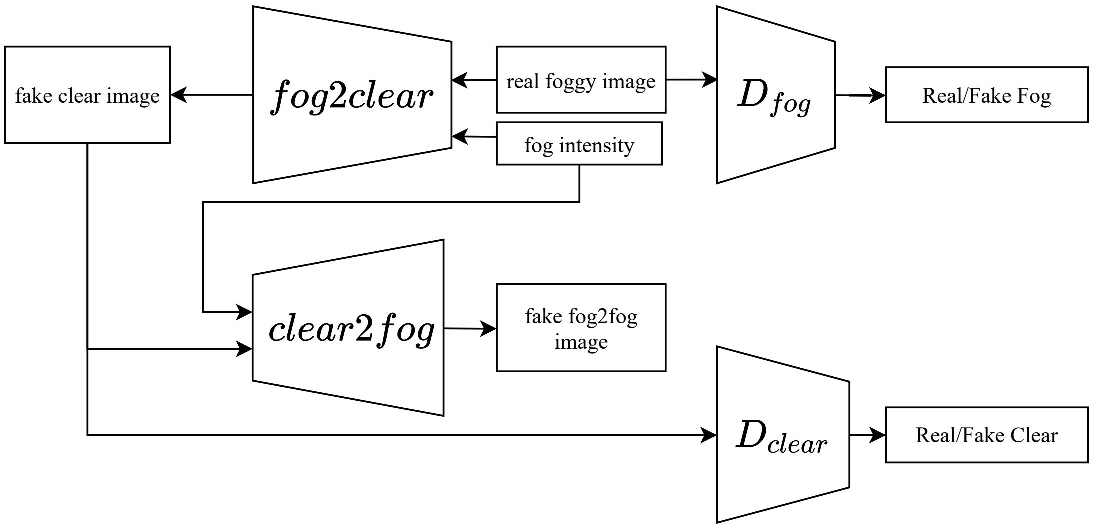
2.4.2 Training Details and Results

Training is done by the `Trainer` class as previously described. In each training epoch, a clear image is converted to a foggy one with a random intensity, then converted back to a clear image (clear2fog2clear cycle), and a foggy image is converted to a clear image then back to a foggy one (fog2clear2fog cycle). When these 2 cycles are executed, the loss values explained in subsection 2.3.2 are calculated for all the generators and dicriminators, gradients are also calculated and applied to update the model's weights. Both cycles are illustrated in Figure 2.8

Due to Colab limitations, it is not possible to run a notebook file for a long time, for this reason, the training process is repeated several times, each time the training takes off where it previously left. That is why the weights and logs are being stored after every epoch. Figure 2.9 is an example of what is being plotted after every epoch during the training process, the generators and discriminators outputs are visualized all together.



(a) Cycle clear2fog2clear



(b) Cycle fog2clear2fog

Figure 2.8: Foggy-CycleGAN Training Phases

As previously mentioned, two versions of Clear to Fog generator are implemented and tested ($clear2fog$ and $clear2fog\text{-}v2$) and thus two versions of Foggy-CycleGAN (Foggy-CycleGAN-v1 and Foggy-CycleGAN-v2) exist. In the following sections, we will visualize the training results for both versions.

Foggy-CycleGAN-v1

In this version, $clear2fog$ is used to generate foggy images out of clear ones, the model is trained for 290 epochs and the whole process took around 4 days. Each epoch lasted for an average of 500 seconds and thus the total effective time was 40 hours. Figure 2.10 shows the loss values for all generators and discriminators after each epoch. What is most important for us is $clear2fog$'s performance, nevertheless, the other losses are also monitored as they all affect each other. We can see that the loss value is not consistently decreasing, but each generator is

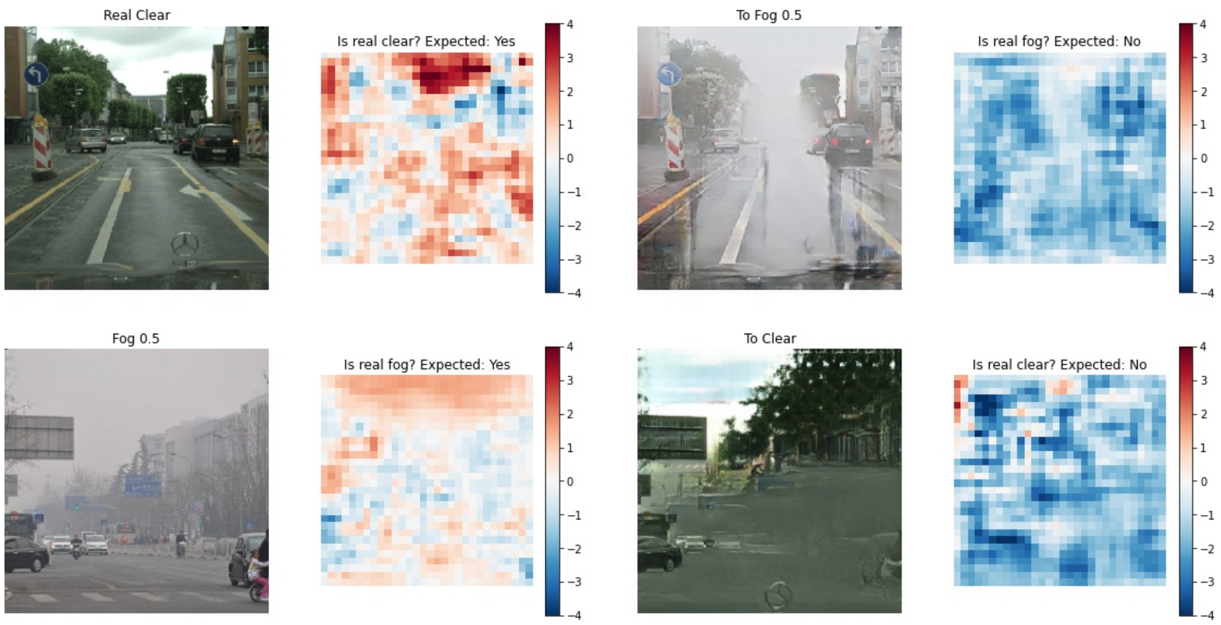
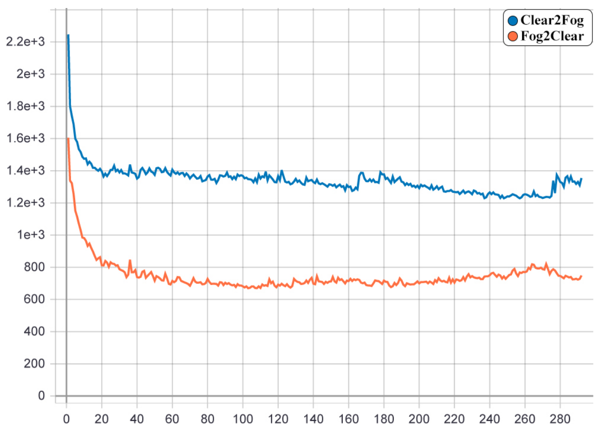
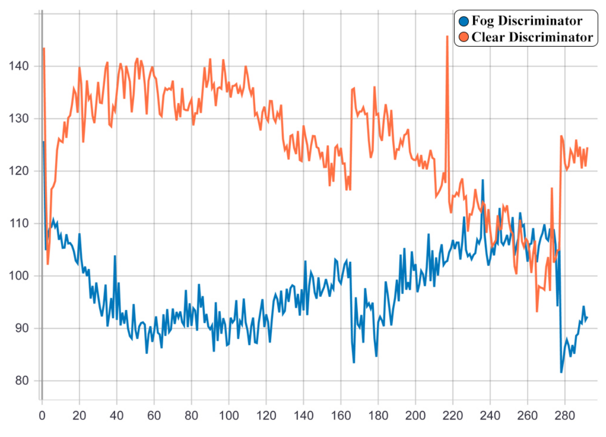

Real Clear
Is real clear? Expected: Yes
To Fog 0.5
Is real fog? Expected: No
Fog 0.5
Is real fog? Expected: Yes
To Clear
Is real clear? Expected: No

Figure 2.9: Visualizing training progress while training

playing a mini-max game with the discriminator. When $clear2fog$'s loss decreases, that means it is getting better at generating foggy images out of clear ones, and thus D_{fog}'s loss will increase. When D_{fog} is trained, its loss decreases as it is getting better at differentiating between real and generated foggy images, making the generator's loss increase. The training was done for 290 epochs, but the final model is the one at the 130th epoch, as the final result seemed to be better.



(a) Generators Loss Values



(b) Discriminators Loss Values

Figure 2.10: Loss values for the Generators and Discriminators of Foggy-CycleGAN-v1 for 290 epochs

As image logs were stored along the training process, we are able to see how the models improved over time. Figure 2.11 shows $clear2fog$'s output for a sample clear image passing an intensity value of 0.5. We can clearly see that the model is not always giving better results as the model is trained more, artifacts start to appear at later epochs.

We can also take a look at $fog2clear$'s output in Figure 2.12. As expected, the model is learning to replace foggy parts of the image with an estimated value of what could be there. The

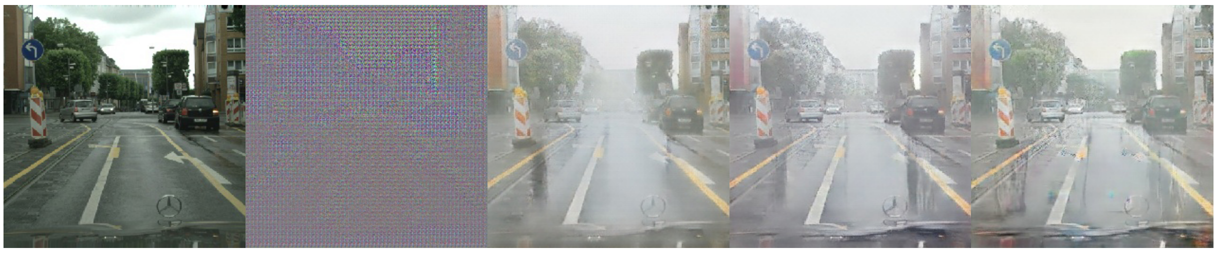


Figure 2.11: Foggy-CycleGAN-v1 epochs outputs for $clear2fog$ generator with required intensity of 0.5

result is clearly very far from the truth, the model is learning how clear images look like, and trying to place trees or buildings in place of white (foggy) parts of the image. Even though the result is not reasonable, it is still valid for our purpose.

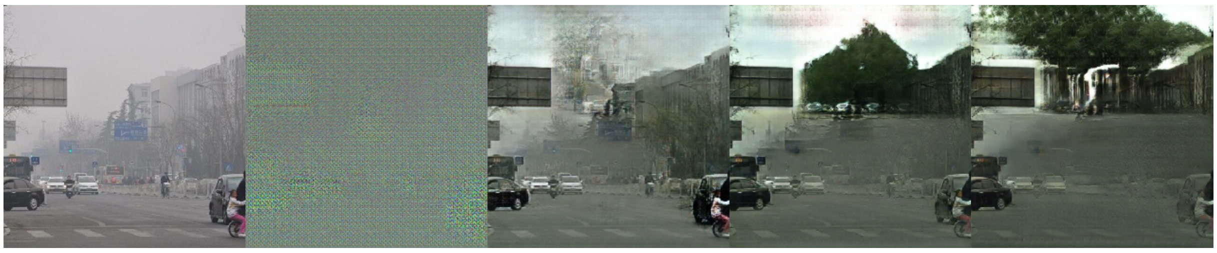
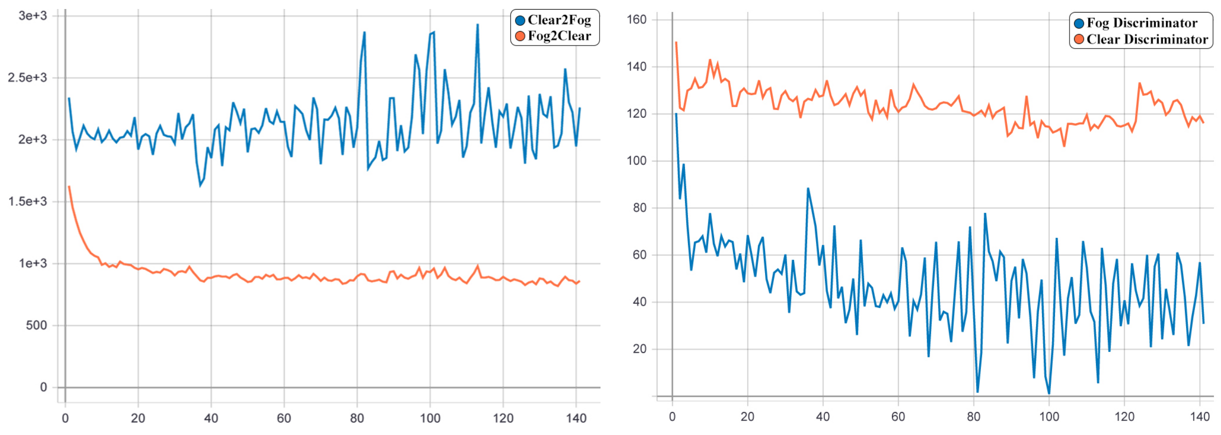


Figure 2.12: Foggy-CycleGAN-v1 epochs outputs for $fog2clear$ generator with input intensity of 0.5

Foggy-CycleGAN-v2

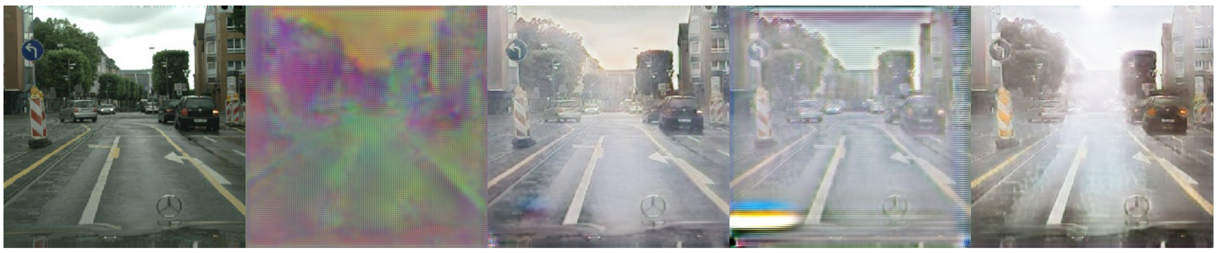
In this version of the model, $clear2fog$-$v2$ was used, while the rest is similar to the previous version. This model was trained for 140 epochs and the training process was significantly slower than the previous one, with an average of 650 seconds per epoch. The model is trained for an effective time of 25 hours over 2 days. Figure 2.13 shows the loss values for all the models, we can see that the loss values for the new generator and its equivalent discriminator (D_{fog}) are noisier than the ones in the previous version.

The outputs of the generators are also plotted during the training, Figure 2.14 and Figure 2.15 show the outputs of $clear2fog$-$v2$ and $fog2clear$ in this version. We notice that the fog in the new model looks smoother. But the image quality got decreased significantly. While no difference in the result of $fog2clear$ is noticed.



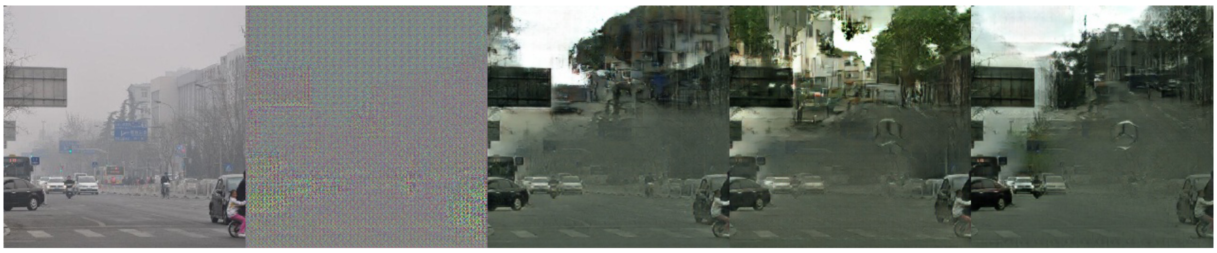
(a) Generators Loss Values (b) Discriminators Loss Values

Figure 2.13: Loss values for the Generators and Discriminators of Foggy-CycleGAN-v2 for 140 epochs



Clear Epoch 0 Epoch 50 Epoch 100 Epoch 140

Figure 2.14: Foggy-CycleGAN-v2 epochs outputs for $clear2fog\text{-}v2$ generator with required intensity of 0.5



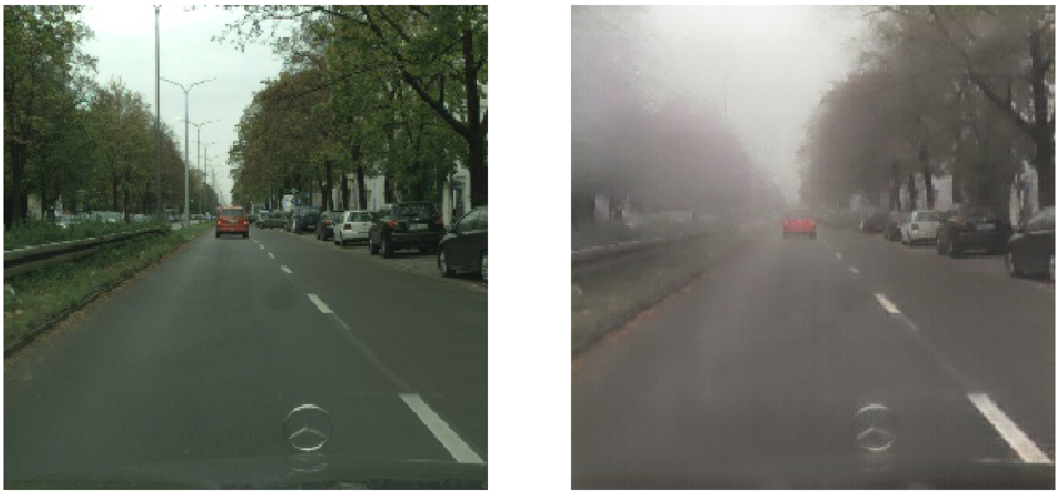
Clear Epoch 0 Epoch 50 Epoch 100 Epoch 140

Figure 2.15: Foggy-CycleGAN-v2 epochs outputs for $fog2clear$ generator with input intensity of 0.5

2.5 Tests and Results

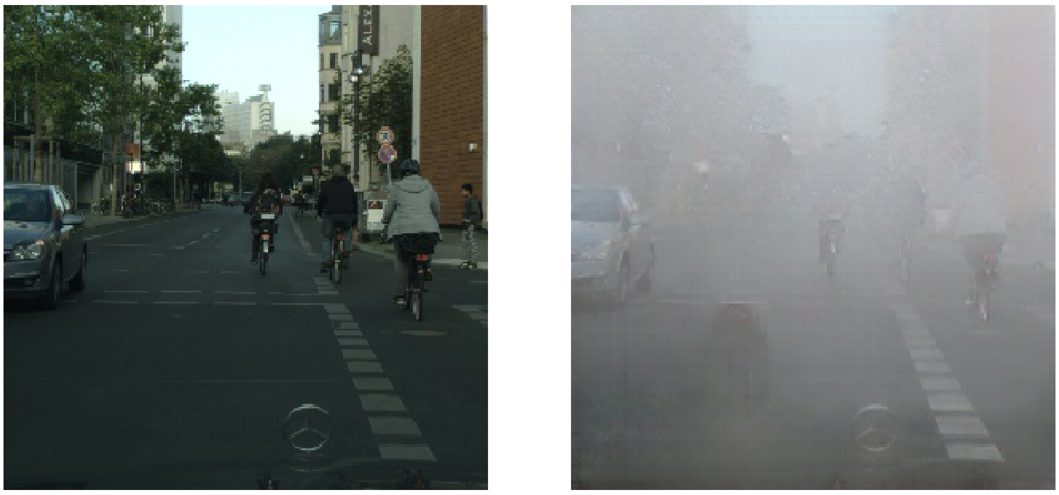
The following results in this section are applied to testing images that were not part of the training dataset. Figures 2.16, 2.17, 2.18 and 2.19 show the result of the first version $clear2fog$. We can see that the results are quite convincing, the generated images looks like a foggy version of the given clear image.

In the Figures 2.20 and 2.21, we can see the output of the second generator $clear2fog\text{-}v2$ on two images. Even though the fog in this version looks smoother, the resolution of the generated



Clear Intensity=0.15

Figure 2.16: $clear2fog$ output - Sample 1



Clear Intensity=0.45

Figure 2.17: $clear2fog$ output - Sample 2

images are lower than the first version, and the result looks less believable. This can be seen in Figure 2.22 that compares between the outputs of the two versions of Foggy-CycleGAN.

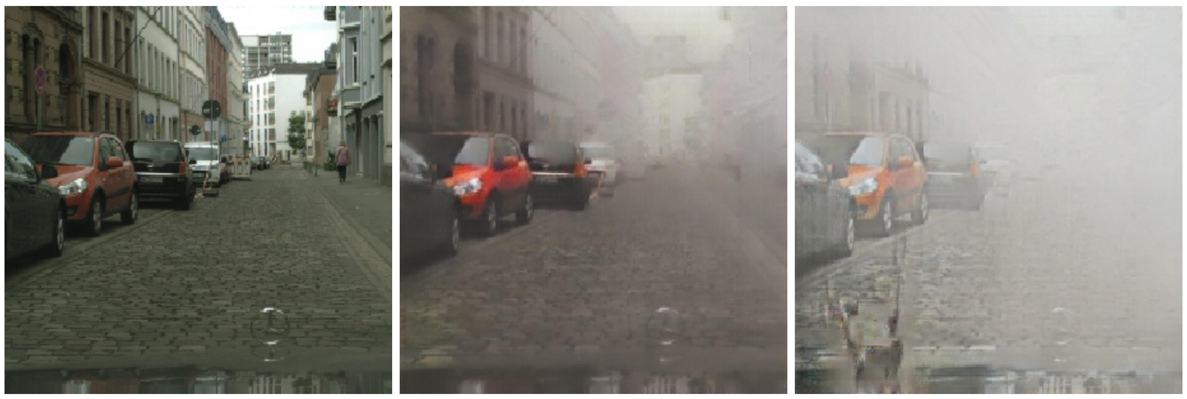


Figure 2.18: $clear2fog$ output - Sample 3

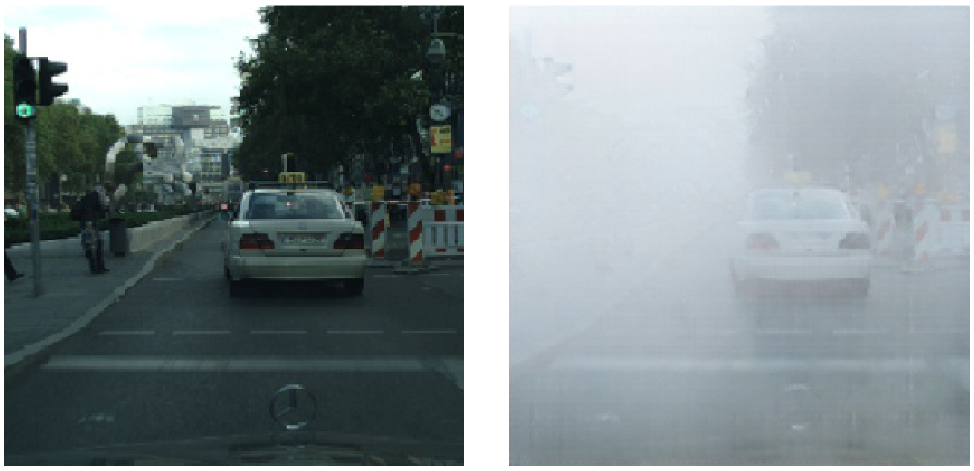


Figure 2.19: $clear2fog$ output - Sample 4

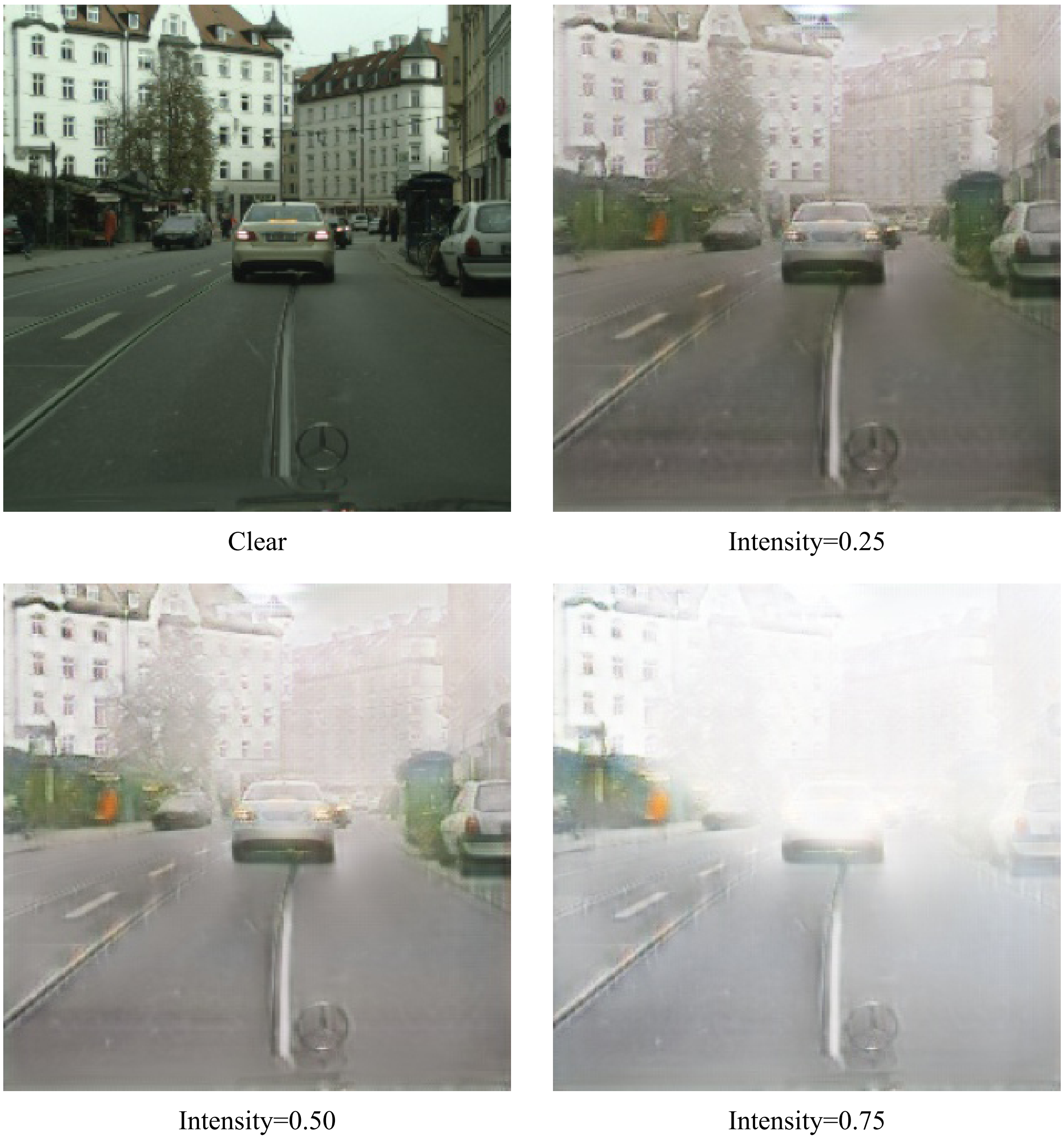


Figure 2.20: $clear2fog$-$v2$ output - Sample 1

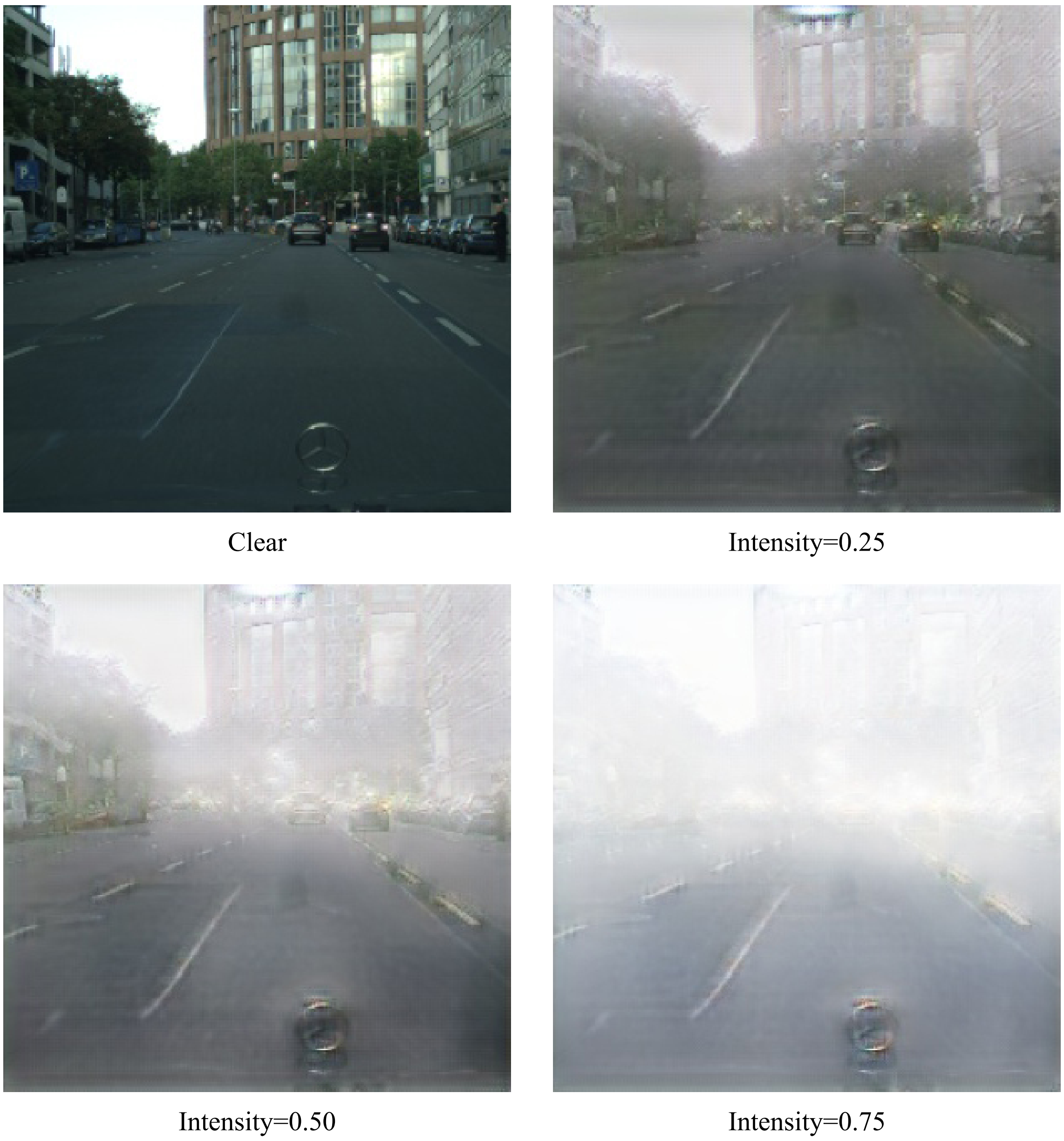


Figure 2.21: $clear2fog$-$v2$ output - Sample 2

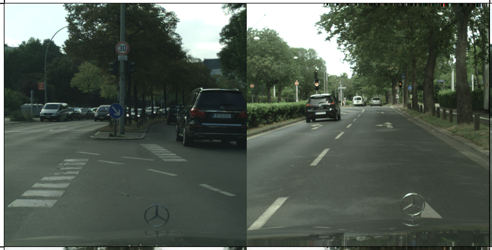
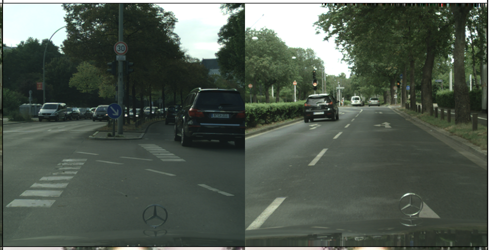
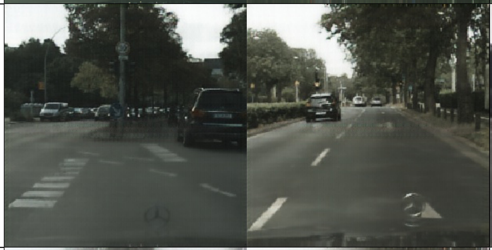
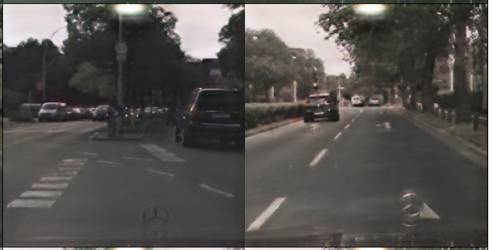
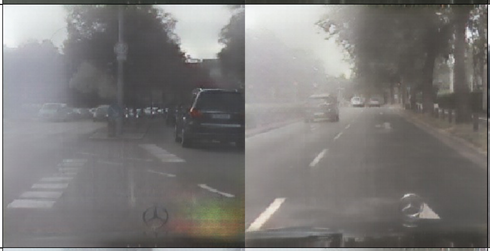
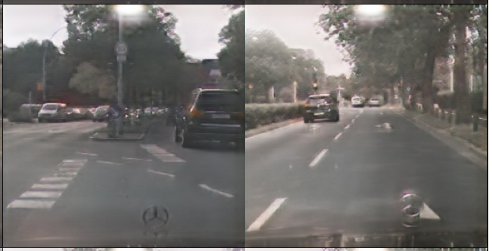
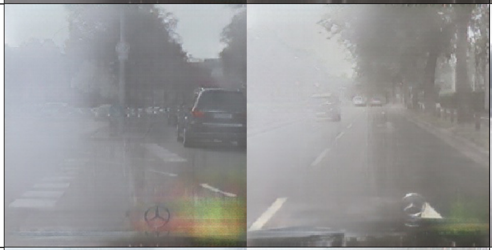
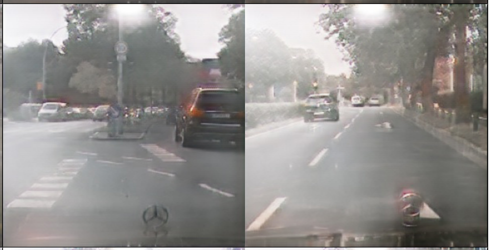
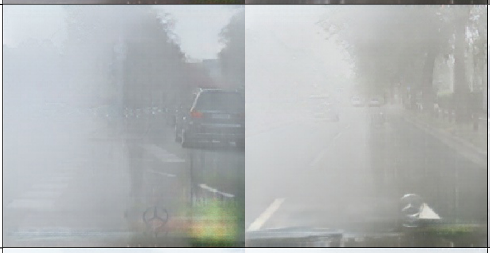
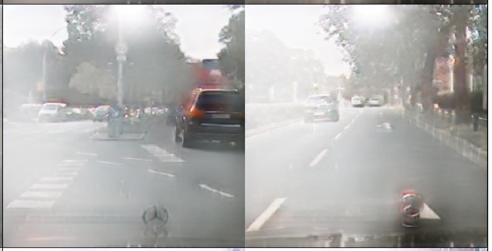
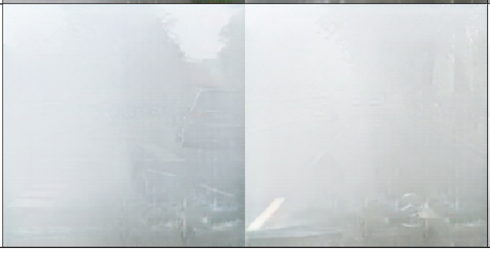
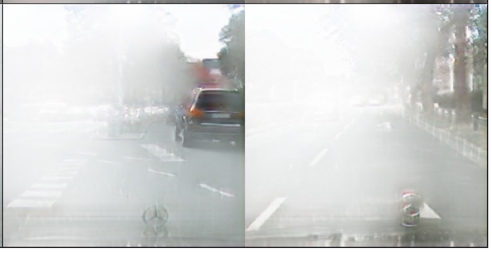
	$clear2fog$	$clear2fog$-$v2$
Clear Input		
Intensity=0		
Intensity=0.2		
Intensity=0.4		
Intensity=0.6		
Intensity=0.8		

Figure 2.22: Comparison between $clear2fog$ and $clear2fog$-$v2$ outputs

Chapter 3

Discussion

3.1 Contribution

Our proposed method provides a way to synthesize fog on clear images using an unpaired dataset. While previous methods used mathematical approaches, mostly based on the depth map of the given image, our method only relies on the information of the fog intensity in a dataset of unpaired the image. Additionally, we present annotations to previously existing datasets where each image is tied to a fog intensity estimation, along with Image Number Annotator tool that was introduced in subsection 2.2.2

3.2 Limitations

Even though the proposed models are generating some reasonable foggy images for lower fog intensities, the results do not look quite convincing all the time. In addition, some artifacts appear in the generated images. The usage of Resize-Convolutional blocks helped to overcome some of these artifacts, but in other types of noise appeared there.

3.2.1 Resolution and Artifacts

The first trained version $clear2fog$ shows some artifacts in the generated images for high intensity values, this is visible in most of the generated images as shown in Figure 3.1.

Using Resize-Conolutional blocks in $clear2fog$-$v2$ helped overcome previous artifacts and make the fog look smoother, on the other hand, other kinds of artifacts appeared, as seen in Figure 3.2b. In addition to that, a clear distortion in the colors and a reduction in the resolution are visible in this version like in Figure 3.2a.

3.2.2 Different Kind of Images

During the dataset collection and annotation, we focused on images that are either taken from inside a vehicle or has similar content, so the model learned to synthesize fog on such images. For that reason, the model does not work well on different kind of images like landscapes or nature images. Figure 3.3 shows an example of this kind of drawback, the fog is not being added in the right part of the image and adding 100% of intensity does not produce a full white image.

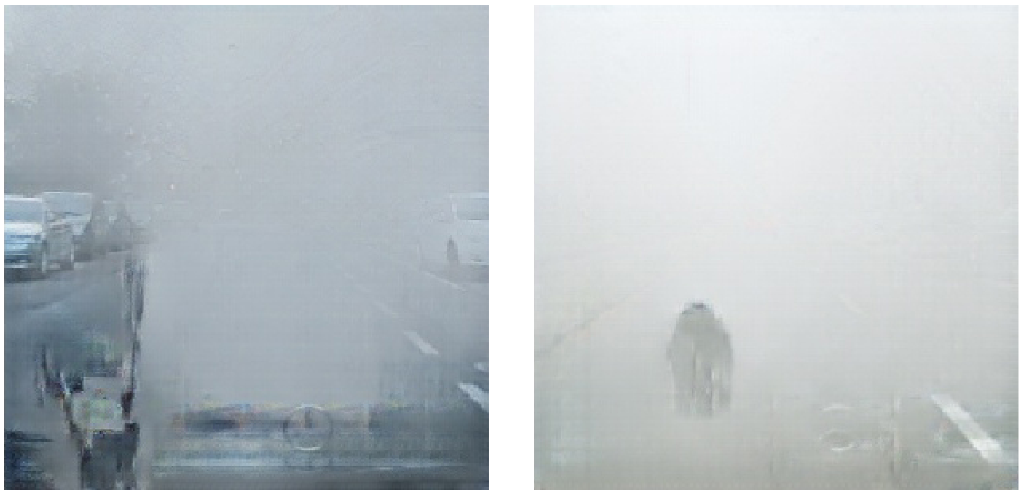

Intensity = 0.6 Intensity = 0.8

Figure 3.1: $clear2fog$ artifacts for high intensities

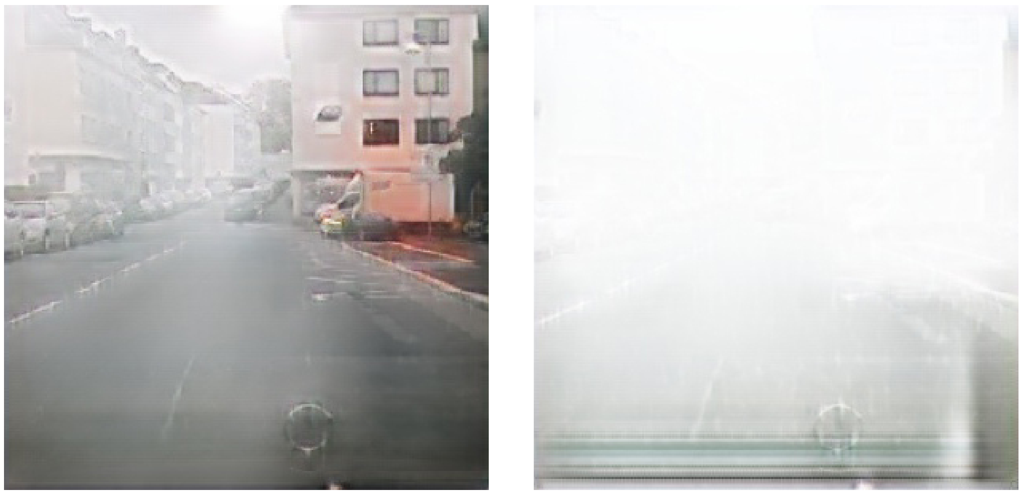

(a) Intensity = 0.5 (b) Intensity = 0.9

Figure 3.2: $clear2fog$-$v2$ artifacts and low resolution

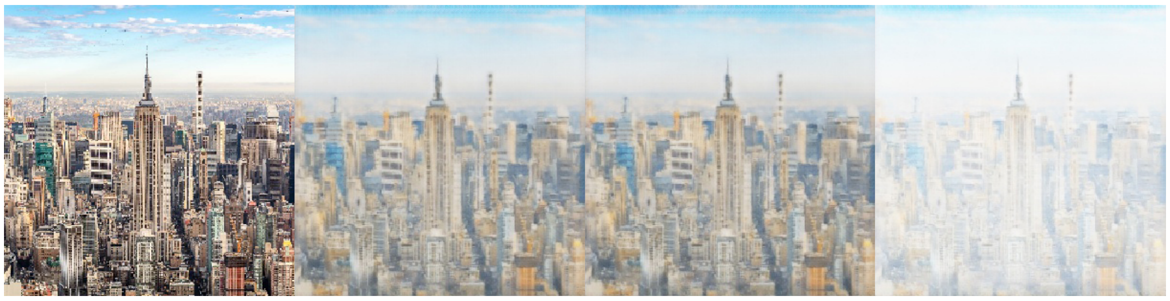

Clear Intensity = 0.5 Intensity = 0.9 Intensity = 1.0

Figure 3.3: $clear2fog$ prediction for different kind of images

3.3 Future Improvements

More improvements can be made to our current model in order to obtain better results and performance, we couldn't apply these improvements either due to the short time, insufficient resources or unavailable data. Some of these thoughts are listed below:

- If a dataset is present, we can train the same model on smoggy images and obtain a smog simulation model. Smog is another weather condition that can distort the visibility and affect autonomous driving systems, being able to generate such images can help improve object detection in case of smog presence.
- Driving during foggy weather can be more dangerous at night. In our model, we focused on day-light images, but if more night foggy images are available, a model can be trained to synthesize fog on clear images taken at night.
- Even though CycleGAN is powerful, but if a paired dataset exists of foggy and non-foggy images, we can produce much better images using the same above structures for $clear2fog$ and D_{fog} and following Pix2pix's idea [21]. It is difficult to obtain such pictures from the nature, but if an installed fixed camera is recording 24/7 footage, we can go through the log history and grab some foggy and non-foggy images from it. Also, if an equipped laboratory is able to generate real fog in a small environment (like a small city), we will be able to collect such dataset.
- Intensity representation proposed in our model can be re-visited; instead of duplicating the same value and adding it as a fourth layer, considering this value as part of the latent vector obtained from the Down-sampling blocks.
- Using the depth information in the images can also be a useful addition that may produce better results if it is available. Such information will allow us to express the fog intensity as a visibility distance.
- If more computing resources are available, having a larger and more diverse dataset will help to train a model that is able to add fog for any kind of images, not just ones taken from a car or that contains one specific type of information like in our case.

3.4 Conclusion

Fog simulation on digital images is a challenging goal to achieve, the reasons for this challenge go from the complexity of applying the mathematical model on real images to the difficulty of obtaining both foggy and clear images of the same content in real life. On the other hand, the availability of a large dataset of both clear and foggy images allowed us to use a generative model, following CycleGAN's approach, to synthesize fog. Instead of going through the trouble of formatting a mathematical formula, calculating the depth data for the clear image's pixels and generating fog using that information, we trained our Foggy-CycleGAN model to figure out how foggy images look like, how clear images look like and how to convert between these two kinds of images keeping the same content.

In our study, we annotated a large number of images giving each an estimated real number that expresses the percentage of fog present in the image. This addition allowed us not only to convert clear images to hazy ones, but also to specify the intensity of fog we are wishing to add to the image.

Our implemented generative models proved to be learning to do the task with reasonable results. Nevertheless, they do not provide the desired outcome for all the kinds of images. In addition, the generated fog does not look smooth for all values of intensities which shows undesired artifacts in some cases. More improvements can be done in a later research to overcome these problems, if more powerful resources are available, a more complex model with a larger dataset can be trained to perform better. Also, if it is possible to generate paired foggy and clear images using special laboratory equipment, it will be possible to train a modified version of our model that will have more realistic outcome with less noise.

In conclusion, we can say that generative models proved to be a good way to synthesize weather conditions on digital images, taking away the trouble of manually building and assessing a mathematical model, and passing that task to the convolutional generative models to accomplish.

Bibliography

[1] Jun-Yan Zhu et al. “Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks”. In: *CoRR* abs/1703.10593 (2017). arXiv: 1703.10593.

[2] Eric Dumont and Viola Cavallo. “Extended Photometric Model of Fog Effects on Road Vision”. In: *Transportation Research Record* 1862 (Jan. 2004), pp. 77–81. DOI: 10.3141/1862-09.

[3] Zhaohui Liu et al. “Analysis of the Influence of Foggy Weather Environment on the Detection Effect of Machine Vision Obstacles”. In: *Sensors* 20 (Jan. 2020), p. 349. DOI: 10.3390/s20020349.

[4] Sahil Dhawan and Jagdish Raheja. “Obstacles Detection in Foggy Environment”. In: Aug. 2013.

[5] Sarthak Katyal et al. “Object Detection in Foggy Conditions by Fusion of Saliency Map and YOLO”. In: Dec. 2018, pp. 154–159. DOI: 10.1109/ICSensT.2018.8603632.

[6] Gurveer Singh and Ashima Singh. “Object Detection in Fog Degraded Images”. In: *International Journal of Computer Science and Information Security (IJCSIS)* 16.8 (2018).

[7] Nan Dong et al. “Adaptive Object Detection and Visibility Improvement in Foggy Image”. In: *Journal of Multimedia* 6 (Feb. 2011), pp. 14–21. DOI: 10.4304/jmm.6.1.14-21.

[8] Christos Sakaridis, Dengxin Dai, and Luc Van Gool. “Semantic Foggy Scene Understanding with Synthetic Data”. In: *CoRR* abs/1708.07819 (2017). arXiv: 1708.07819.

[9] Kyeong Jeong and Byung Song. “Fog Detection and Fog Synthesis for Effective Quantitative Evaluation of Fog–detection-and-removal Algorithms”. In: *IEIE Transactions on Smart Processing & Computing* 7 (Oct. 2018), pp. 350–360. DOI: 10.5573/IEIESPC.2018.7.5.350.

[10] C. Sun et al. “An algorithm of imaging simulation of fog with different visibility”. In: *2015 IEEE International Conference on Information and Automation*. Aug. 2015, pp. 1607–1611. DOI: 10.1109/ICInfA.2015.7279542.

[11] Rachid Belaroussi and Dominique Gruyer. “Impact of Reduced Visibility from Fog on Traffic Sign Detection”. In: June 2014, pp. 1302–1306. ISBN: 978-1-4799-3638-0. DOI: 10.1109/IVS.2014.6856535.

[12] Thomas Wiesemann and Xiaoyi Jiang. “Fog Augmentation of Road Images for Performance Analysis of Traffic Sign Detection Algorithms”. In: *Advanced Concepts for Intelligent Vision Systems*. Ed. by Jacques Blanc-Talon et al. Cham: Springer International Publishing, 2016, pp. 685–697. ISBN: 978-3-319-48680-2.

[13] Fan Guo, Jin Tang, and Xiaoming Xiao. “Foggy Scene Rendering Based on Transmission Map Estimation”. In: *International Journal of Computer Games Technology* 2014 (Oct. 2014). DOI: 10.1155/2014/308629.

[14] Hochang Lee, J. R. Jang, and Kyunghyun Yoon. “Fog Rendering Using Distance-Altitude Scattering Model on 2D Images”. In: 2012.

[15] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661.

[16] Alec Radford, Luke Metz, and Soumith Chintala. “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”. In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016.

[17] Tero Karras et al. “Progressive Growing of GANs for Improved Quality, Stability, and Variation”. In: *CoRR* abs/1710.10196 (2017). arXiv: 1710.10196.

[18] Tero Karras, Samuli Laine, and Timo Aila. “A style-based generator architecture for generative adversarial networks”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 4401–4410.

[19] Tero Karras et al. “Analyzing and Improving the Image Quality of StyleGAN”. In: *arXiv preprint arXiv:1912.04958* (2019).

[20] Mehdi Mirza and Simon Osindero. “Conditional Generative Adversarial Nets”. In: *CoRR* abs/1411.1784 (2014). arXiv: 1411.1784.

[21] Phillip Isola et al. “Image-To-Image Translation With Conditional Adversarial Networks”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. July 2017.

[22] David Bau et al. “Semantic Photo Manipulation with a Generative Image Prior”. In: *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)* 38.4 (2019).

[23] Ting-Chun Wang et al. “Video-to-Video Synthesis”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2018.

[24] Aidan Clark, Jeff Donahue, and Karen Simonyan. “Efficient Video Generation on Complex Datasets”. In: *CoRR* abs/1907.06571 (2019). arXiv: 1907.06571.

[25] Andrew Brock, Jeff Donahue, and Karen Simonyan. “Large Scale GAN Training for High Fidelity Natural Image Synthesis”. In: *CoRR* abs/1809.11096 (2018). arXiv: 1809.11096.

[26] J. Thies et al. “Real-time Expression Transfer for Facial Reenactment”. In: *ACM Transactions on Graphics (TOG)* 34.6 (2015).

[27] J. Thies et al. “Face2Face: Real-time Face Capture and Reenactment of RGB Videos”. In: *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*. 2016.

[28] Jiajun Wu et al. “Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling”. In: *CoRR* abs/1610.07584 (2016). arXiv: 1610.07584.

[29] Chris Donahue, Julian J. McAuley, and Miller S. Puckette. “Synthesizing Audio with Generative Adversarial Networks”. In: *CoRR* abs/1802.04208 (2018). arXiv: 1802.04208.

[30] Aäron van den Oord et al. “WaveNet: A Generative Model for Raw Audio”. In: *CoRR* abs/1609.03499 (2016). arXiv: 1609.03499.

[31] Marius Cordts et al. “The Cityscapes Dataset for Semantic Urban Scene Understanding”. In: *CoRR* abs/1604.01685 (2016). arXiv: 1604.01685.

[32] Deniz Engin, Anil Genç, and Hazim Kemal Ekenel. “Cycle-Dehaze: Enhanced CycleGAN for Single Image Dehazing”. In: *CoRR* abs/1805.05308 (2018). arXiv: 1805.05308.

[33] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).

[34] Wei Liu et al. “End-to-End Single Image Fog Removal using Enhanced Cycle Consistent Adversarial Networks”. In: *CoRR* abs/1902.01374 (2019). arXiv: 1902.01374.

[35] Boyi Li et al. “Benchmarking Single-Image Dehazing and Beyond”. In: *IEEE Transactions on Image Processing* 28.1 (2019), pp. 492–505.

[36] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. “U-Net: Convolutional Networks for Biomedical Image Segmentation”. In: *CoRR* abs/1505.04597 (2015). arXiv: 1505.04597.

[37] Dmitry Ulyanov, Andrea Vedaldi, and Victor S. Lempitsky. “Instance Normalization: The Missing Ingredient for Fast Stylization”. In: *CoRR* abs/1607.08022 (2016). arXiv: 1607.08022.

[38] Tensorflow. *Tensorflow implementation of CycleGAN*. URL: https://www.tensorflow.org/tutorials/generative/cyclegan.

[39] Augustus Odena, Vincent Dumoulin, and Chris Olah. “Deconvolution and Checkerboard Artifacts”. In: *Distill* (2016). DOI: 10.23915/distill.00003.

[40] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015.